TECHNIQUES OF PROGRAM STRUCTURE AND DESIGN

Edward Yourdon
President
YOURDON Inc.

Prentice-Hall, Inc., Englewood Cliffs, New Jersey
1975

Э.ЙОДАН

СТРУИТУРНОЕ ПРОЕКТИРОВАНИЕ И КОНСТРУИРОВАНИЕ ПРОГРАММ

Перевод с английского
В. В. ФРОЛОВА и
Л. А. ТЕПЛИЦКОГО

под редакцией
Л. Н. КОРОЛЕВА

ИЗДАТЕЛЬСТВО «МИР»
МОСКВА 1979

УДК 681.142.2

На современном уровне обсуждены принципы и методы структурного программирования, позволяющие получать надежно работающие и хорошо документированные программы. Изложены конструктивные стратегии разработки относительно просто реализуемых и обслуживаемых модульных программ. Подробно разобран так называемый нисходящий подход к построению программ и его варианты. Рассмотрены особенности и стиль модульного и структурного программирования, методы испытания программ и возможные пути их отладки.

Книга рассчитана на программистов—в том числе начинающих— **и** на руководителей групп программного обеспечения. Благодаря инженерной направленности она представляет также несомненный интерес для широкого круга специалистов по вычислительной технике.

Редакция литературы по новой технике

© 1975 Prentice-Hal!, Inc.
© Перевод на русский язык, **«Мир», 1979**

ОТ РЕДАКТОРА РУССКОГО ПЕРЕВОДА

Проблемы организации труда программистских коллективов, а также проблемы разработки больших и сложных программных систем приобрели в последние годы чрезвычайную актуальность. И это не удивительно, поскольку уже многие годы наблюдается экспоненциальный рост стоимости программного обеспечения ЭВМ. В настоящее время затраты на разработку программ превосходят почти вдвое затраты на аппаратные средства. Хорошо известно, что задержки в создании необходимых программ или их некачественное составление приводят к большим потерям в эффективности использования ЭВМ. В результате этого отдаляются сроки окупаемости дорогостоящих установок и сдерживаются темпы развития ряда научно-технических отраслей, ориентирующихся на применениеЭВМ.

По некоторым прогнозам через десятилетие затраты на математическое обеспечение будут в 10 раз превышать затраты на вычислительное оборудование. Поэтому все большее и большее внимание и в теоретическом, и в практическом плане уделяется вопросам разработки инструментальных средств, методик и технологических комплексов, ускоряющих процесс проектирования и отладки программ, и их ввода в эксплуатацию.

Большой отклик в среде специалистов по обработке данных нашли идеи нисходящего программирования, модульного и структурного программирования, а также создания систем построения трансляторов. В области теории и в СССР, и за рубежом публикуется значительное число работ по автоматическому доказательству правильности программ, работ по формализации правил многоуровневого программирования, по расслоенному программированию, которые в конечном итоге нацелены на решение основной задачи — быстрого и правильного построения программ. Почти все специалисты в настоящее время соглашаются с тем, что к созданию программ нужно относиться так же, как к созданию любого другого материального продукта, что программирование следует ставить на промышленную основу и рассматривать программный продукт с точки зрения таких его показателей, как «технологичность», «качество», «эффективность», «документированность» и даже «надежность». В применении к про-

граммированию эти показатели приобретают вполне конкретный смысл, даже несмотря на то, что программный продукт значительно отличается от серийных материальных изделий промышленных предприятий.

Сегодня имеется сравнительно мало литературы, суммирующей опыт организации процесса программирования и его «технологических» циклов, а также данные анализа ошибок, трудностей и скрытых опасностей. И практически отсутствуют книги, доходчиво излагающие проблематику технологии программирования. После «Мифического человеко-месяца» Брукса Ч, книга Йодана — это вторая книга, посвященная организационным и технологическим проблемам программирования, которая адресована широкому кругу читателей. В этой книге больше внимания уделяется вопросам технологии программирования, нежели вопросам организации программистских коллективов. Многочисленные рекомендации Йодана, повторяющиеся при освещении разных аспектов проблемы хорошо организованного программирования, будут полезны начинающим программистам. Опытные программисты, прочитав эту книгу, возможно, не обнаружат в ней особых откровений, однако найдут много созвучного их опыту. Большое число сведений, которые можно почерпнуть из книги, будет полезно руководителям программистских коллективов, хотя, разумеется, не все советы, рецепты и рекомендации автора книги они безаппеляционно примут.

Знакомясь с книгой, следует помнить, что она рассчитана на широкий круг читателей и принадлежит в значительной мере к категории научно-просветительной литературы. Тем не менее даже не претендующее на большую глубину обсуждение ряда таких вопросов, какструктурноепрограммирование иформальноедоказательство правильности построения программ, следует безусловно приветствовать, поскольку оно может стимулировать интерес к этой важной проблематике.

Книга Йодана написана на основе лекций и семинарских занятий, проведенных автором с программистами разного уровня квалификации в целом ряде стран. Это, естественно, не могло не сказаться на принятом стиле изложения, который носит характер свободной беседы. Конечно, всегда найдутся критики, которые выскажут сомнения в уместности такого стиля для подобного издания. Однако нужно признать, что именно благодаря ему многие разделы книги читаются с увлечением и воспринимаются эмоционально, возбуждая безусловный интерес к затронутым проблемам.

проф. Л. Н. Королев

1) Brooks F. B., The Mythical Man-Month and Other Essays on Software Engineering. Prepublication Draft, University of North Carolina, Chapel Hill, 1972.

Посвящается Дженни

ПРЕДИСЛОВИЕ 1)

Мы строим системы так же, как братья Райт строили свои самолеты: создаем всю систему целиком, запускаем ее — пусть онаразвалится! — и начинаем все сначала.

проф. Грехем
Software Engineering, p. 17.

Безусловно, 99% вычислительных машин работают вполне сносно, это правда. Существуют тысячи солидных вычислительных систем, ориентированных на использование Фортрана и включающих множество различных ЭВМ, существует масса систем обработки данных, функционирующих очень надежно; все это так! Вопрос, которым мы озабочены, имеет огромное значение и касается предельных возможностей наших взаимоотношений с ЭВМ.

проф. Бакстон
Software Engineering, p. 119.

Я думаю, это неизбежно, что люди программируют плохо и что так будет и впредь. Обучение не приведет к заметному улучшению дела. Использование специализированных языков не поможет, так как люди всегда нарушают их правила. Нам придется просто привыкнуть к этому.

проф. Перлис
Software Engineering Techniques, p. 33.

Существует глубокая пропасть между притязаниями и реальными достижениями в области математического обеспечения. Эта пропасть ощущается в разных плоскостях: между обещаниями пользователям и характеристиками математического обеспечения, между тем, что принципиально достижимо, и тем, что мы можем получить сегодня, между оценками стоимости математического обеспечения и фактическими затратами на него. И эта пропасть углубляется в такое время, когда последствия отказов математического обеспечения становятся во всех отношениях все более и более серьезными. Особеннотревожит

[1]) Цитаты, служащие своеобразным эпиграфом, взяты из сборника Software Engineering (ред. Наури Ренделл), сектор научных исследований НАТО, Брюссель 39, Бельгия, январь 1969 г., и сборника Software Engineering Techniques (ред. Бакстон и Ренделл), сектор научных исследований НАТО, Брюссель 39, Бельгия, апрель 1970 г.

кажущаяся неизбежной ненадежность больших систем математического обеспечения в связи с тем, что отказы в таких усовершенствованных системах могут иметь жизненно важное значение не только для отдельных индивидуумов, но и для безопасности транспортных средств с сотнями людей и в конечном счете для безопасности целых наций.

д-р Дэвид,
Фрэзер
Software Engineering, p. 120.

* * *

В начале 1970 г. я допустил большую ошибку, написав серию заметок для семинара, который назывался СОВРЕМЕННЫЕ МЕТОДЫ ПРОГРАММИРОВАНИЯ. Я говорю «ошибку», так как вскоре установил, что я почти ничего не знаю о современном состоянии того колдовства, которое мы называем программированием, несмотря на то, что в этой области мне охотно предоставляли работу уже в течение нескольких лет. Но я не отступил: 80 страниц почти бессвязной писанины, пройдя через десять значительных переработок, разрослись до 900 машинописных страниц. Иногда во время этой работы здравый смысл подсказывал мне, чтобы я выбросил все десять вариантов рукописи и перестал навязывать учащимся свои идеи; к сожалению, гордость и обыкновенное самолюбие взяли верх. Бедные слушатели, которым пришлось обучаться в этот период! Почти 3000 программистов из 12 стран прощали мне мои ошибки в программах и не придавали значения некоторым из моих серьезных заблуждений, и важно то, что все они прилежно сообщали, что знали о программировании сами — доверие, которое нужно прочувствовать, чтобы оценить по достоинству.

В этот период меня особенно интересовали перемены, которые наблюдались в области вычислительных средств. Появление таких выдающихся авторитетов, как Дейкстра, Вирт и Вейнберг, вселяло надежду, что наступит время, когда программистов будут обучать написанию хороших программ с самого начала.

Если эти надежды оправдаются, то некоторые предосторожности и соображения, обсуждаемые в этой книге, могут оказаться ненужными. Свои семинары и эту книгу я строю исходя из того предположения, что обучающийся обладает некоторыми основами знаний в области программирования, но совершенно не знаком с идеями «хорошего» программирования. Я считаю такое предположение вполне оправданным, имея в виду подавляющее большинство программистов, работающих в промышленности. До тех пор, пока в основных курсах программирования наибольшее внимание уделяется правилам кодирования на том или ином алгоритмическом языке (как это обычно наблюдается на курсах Фортрана и Кобола),

такое предположение остается справедливым. Очень обнадеживает то, что в настоящее время заметна тенденция включать в университетские курсы для студентов основы искусства хорошего программирования.

Итак, предметом изучения настоящей книги является хорошее программирование. Как подсказывает мой опыт, бывает очень трудно объяснить человеку, как писать хорошую программу, если он (или она) упорно отрицает мой взгляд на то, что называть хорошей программой. Поэтому гл. 1 книги посвящена обсуждению характерных особенностей хорошей программы. Должен признать, что эта глава адресована опытному программисту; первокурсник колледжа вряд ли имеет ясное представление об относительной важности таких свойств программы, как простота сопровождения, гибкость и эффективность.

Далее логически следует вопрос: как проектировать хорошие программы? В гл. 2 ответ на этот вопрос дается изложением идеологии *нисходящего проектирования,* или проектирования сверху вниз. Кажется несколько противоречивым тот порядок, в котором рассматриваются эти разделы. Многие считают, что сначала следует изложить основные идеи структурного программирования, после чего программист будет подготовлен к восприятию принципов нисходящего проектирования. Возможно, что это и так; я сам, сталкиваясь с группой слишком нетерпеливых программистов, считаю полезным начать семинар с конкретных понятий структурного программирования, переходя далее к более абстрактным идеям нисходящего проектирования. И тем не менее с логической точки зрения представляется более разумным изучать сначала *проектирование* программ, а затем — правила их написания.

Гл. 3 посвящена обсуждению модульного программирования. Она служит естественным переходом от абстрактных представлений нисходящего проектирования к более подробному обсуждению структурного программирования. Очень многим кажется, что модульное программирование явилось предвестником весьма модного в наше время структурного программирования. Я нахожу занятным то, что в 1970 г., когда я начинал писать эти заметки, многие программисты считали модульное программирование радикально новым направлением; в наше время структурного программирования модульное программирование уже относят к прошлому. Как ни странно, но весьма вероятно, что именно эта область применения принципов модульности будет характеризоваться наибольшими успехами в ближайшие несколько лет. Возможно, что статья Констентайна «Структурное проектирование» в майском номере журнала IBM Systems Journal за 1974 г. возродит интерес к модульному программированию и сыграет ту же роль, какую в свое время в области структурного программирования сыграла статья Бейкера в январском номере этого журнала за 1972 г.

Безусловно, всякий современный учебник программирования должен содержать обстоятельное изложение методов структурного программирования; гл. 4 посвящена подробному обсуждению этих вопросов. Один из разделов этой главы заслуживает особого упоминания: в разд. 4.3.3 обсуждаются методы преобразования неструктурированных программ в структурированные программы. Многие считают преувеличенным то внимание, которое я уделяю этим методам, но я беру на себя смелость утверждать, что полезность материала этого раздела зависит от индивидуального опыта обучающегося. Новичок, еще не успевший приобрести привычку запутывать свои программы многочисленными передачами управления по оператору GO TO, вероятно, совсем не нуждается в методах, излагаемых в разд. 4.3.3; знакомство с ними могло бы оказаться для него даже вредным! Опытному же программисту эти методы крайне необходимы, поскольку они помогают обратить неструктурное *мышление* в структурное. Довольно наивно ссылаться на то, что сейчас в университетах с самого начала обучают структурному программированию. Эта аргументация аналогична логике, требующей немедленно отказаться от программирования на Фортране только потому, что в нем не предусмотрены форматы, необходимые для удобного представления структурированных программ. В жизни все обстоит иначе. Хорошо это или плохо, но еще несколько лет люди будут по-прежнему программировать на Фортране, пока мы не сможем предложить им более развитые языки. А до того времени с практической точки зрения весьма важно обучить программистов приближенной реализации идей структурного программирования в пределах возможностей Фортрана. Кроме того, сегодня в мире насчитываются *сотни тысяч* опытных программистов, и каждый из них, если ему не доведется познакомиться с методами преобразования неструктурированных программ в структированные, еще двадцать лет будет писать свои запутанные программы. Если мы будем ждать обученных по-новому выпускников университетов, которые придут и поправят дело чудодейственным средством структурного программирования, большинство существующих вычислительных систем придет в упадок.

Должен признаться, что после трех глав проповеди на тему структурного программирования я на какое-то время выдохся. Гл. 5 и 6, посвященные рассмотрению вопросов стиля и выбора надежных схем программирования, заслуживают, по-видимому, большего внимания. В качестве оправдания могу высказать следующее: если программист последовательно придерживается принципов, изложенных в гл. 1—4,то по здравому смыслу он логически приходит к положениям, содержащимся в гл. 5 и 6. Болеевескимдоводомможет служить то, что подробному анализу вопросов стиля в программировании следовало бы посвятить отдельную книгу. Я настоятельнорекомендую в этой связи две книги:Schneiderman, Kreitzberg,

The Elements of FORTRAN Style, Harcourt, Brace, Jovanovich, 1971 и Kernighan, Plauger, The Elements of Programming Style, McGraw-Hill, 1974.

По нескольким причинам мне стало ясно, что обсуждение методов нисходящего проектирования, а такжемодульного и структурного программирования необходимо связать с вопросами тестирования и отладки. Так появились гл. 7 и 8. Я и теперь настаиваю на том, что следует делать различие между тестированием и отладкой, хотя многие из моих слушателей считают, что я усложняю дело. Должен сознаться, что, закончив главу, посвященную тестированию, я испытал некоторое чувство безнадежности: мне кажется, что большинство программистов в действительности не представляют, как нужно тестировать программы. Большинство ученых, работающих в области программирования, вероятно, согласятся с тем, что прежде, чем мы достигнем в тестировании того уровня упорядоченности, который мы имеем в структурном программировании, предстоит сделать еще очень многое. Может быть, нам потребуется «структурное тестирование»?

Если можно говорить о структурном тестировании, то почему бы не сказать о структурной отладке? Основная цель гл. 8 состоит в описании некоторых стратегий отладки программ, весьма отличающихся от традиционных методов трассировки и дампинга. Я всегда считал, что эти стратегии предполагают определенное искусство отладки, и если программист затрудняется в их применении, то причину следует искать в отсутствии некоторых врожденных способностей, присущих тонким аналитикам программ. Может быть, это слишком простое объяснение: как мне рассказали друзья из мельбурнской фирмы Shell Oil, в этой организации программистам читаются общие курсы из области принятия решения, имеющие целью повышение их мастерства в отладке программ. Возможно, это и есть начало структурной отладки.

В конце этой книги я поместил четыре больших упражнения по программированию, на которых могут быть опробованы основные принципы проектирования программ. Задачи, помещенные в приложенияхА иБ,предназначаются для решения группой программистов.оптимальный состав группы—три-четырепрограммиста. Задачи приложений В и Г не так громоздки и могут быть решены каждым программистом индивидуально.

* * *

Не забывая тысяч слушателей, прочитавших рукопись этой книги и внесших свои улучшения и исправления, я хотел бы выразить особую благодарность Кернигану из фирмы Bell Laboratories и Ниили из Канзасского университета за рецензирование книги. Я благодарен также своим коллегам: Плогеру, Сарсону и Эбботу,

которые использовали материал книги в своих курсах и высказали рекомендации по улучшению текста рукописи. Готовя книгу к выпуску, Линн Садковски не теряя самообладания, неизменно разыскивал меня в отдаленном районе земного шара, чтобы я успел просмотреть критические места в гранках; Уэнди Икин оказала мне неоценимую помощь в редактировании рукописи и чтении корректур, добившись того, что текст обрел черты сносного английского языка.

Наконец, появлением этой книги я обязан Сэму, который верил в меня больше других.

ГЛАВА 1

ОТЛИЧИТЕЛЬНЫЕ ОСОБЕННОСТИ „ХОРОШЕЙ" ПРОГРАММЫ ДЛЯ ЭВМ[1)]

Понятие программы является по существу центральным понятием в области математического обеспечения, однако до сих пор мы не имеем для него достаточно общего и конструктивного определения. В наиболее распространенном определении программы как последовательности команд не учитывается значение исходных данных к программе. Более точным является определение программы как совокупности преобразований и других связей, которые соотносят структуры данных структурам используемых носителей информации. Такое определение по крайней мере предполагает необходимость разбиения задачи проектирования программы на подзадачи выбора подходящих структур данных и носителей, а также задания соответствующих операторов. Этим определением подчеркивается необходимость рассмотрения свойств данных независимо от особенностей носителей (записи, слова, разделы и т. д.).

Колене
Software Engineering, p. 50.

Не существует теории, которая позволяла бы рассчитывать предельные размеры, характеристики и сложность математического обеспечения. Во многих случаях у нас даже нет логически строгих методов определения того, какую задачу должна решать данная система математического обеспечения и каким образом она должна ее решать.

Дэвид
Software Engineering, p. 69.

Одна из важнейших задач в области разработки систем математического обеспечения состоит в том, чтобы определить критерий прогресса в процессе разработки и найти способ его измерения. Пока нам кажется ясным только одно. Это то, что создание программы не всегда представляется прямолинейным продвижением к цели, где каждая операция объединения элементов в целое представляет собой явный шаг вперед, и что конечный результат разработки не всегда есть простое объединение многочисленных подсистем.

Фрузер
Software Engineering, p. 86.

[1)] Цитаты в начале главы взяты из сборника Software Engineering (ред. Наур и Ренделл), сектор научных исследований НАТО, Брюссель 39, Бельгия, январь 1969 г., и сборника Software Engineering Techniques (ред. Бакстон и Ренделл), сектор научных исследований НАТО, Брюссель 39, Бельгия, апрель 1970 г.

Руководители разработками в области программирования будут заслуженно пользоваться дурной репутацией за свои неэффективные решения вопросов планирования и оценки стоимости до тех пор, пока мы не достигнем более полного понимания самого процесса проектирования программ.

Колене
Software Engineering, p. 123.

Всякая большая программа на протяжении своего жизненного цикла существует в нескольких различных вариантах, поэтому при создании большой программы мы имеем дело не с какой-нибудь единственной программой, а с целым семейством взаимосвязанных программ, включающим альтернативные программы для решения одной и той же задачи и (или) подобные программы для решения подобных задач. Таким образом, всякую программу нужно рассматривать и проектировать как элемент семейства; ее следует конструировать из компонентов таким образом, чтобы в различных элементах семейства обеспечивалась правильная работа не только общих компонентов, но и общих подсистем, сконструированных из этих компонентов.

Дейкстра
Software Engineering Techniques, p. 31.

1.0. Введение

На протяжении всей этой книги мы предполагаем, что вам уже известны основные элементы вычислительных машин, операционных систем и алгоритмических языков; теперь вы намерены уделить внимание более тонким вопросам программирования.Какие разделы программирования вы предпочли бы изучить прежде всего? Может быть вас интересуют списковые структуры? Или динамическое распределение памяти? Или таблицы решений? А может быть, алгоритмы сортировки и поиска?

Если вы выбрали *любой* из названных разделов, то вам, возможно, свойствен недостаток, характерный почти для всех программистов,— крайнее пристрастие к *технике* программирования. Можете быть уверены, что в свое время мы обсудим некоторые важные методы программирования, но лишь после того, как условимся об определенных *принципах* в программировании.

Принципиальные вопросы обычно не пользуются популярностью у слушателей-программистов; эти вопросы кажутся им слишком неопределенными и общими, а наши программисты предпочитают тратить свое время на обсуждение более «практических» и «полезных» вещей. В то же время следует напомнить, что во многих случаях программисты оказываются людьми упрямыми, непрактичными и несговорчивыми: им часто кажется, что их основным назначением является изобретение новых умных алгоритмов, а не выполнение полезной работы.

Не желая быть излишне грубым, я все-таки должен напомнить об одном важном обстоятельстве: *как программист, вы всегда бу-*

дете работать на того, кто вас нанимает. Если вы не обладаете чрезмерным богатством и не отличаетесь эксцентричностью, вы никогда не можете позволить себе роскошь пользоваться вычислительной машиной у себя дома, и если вы не собираетесь оставаться вечным студентом, вы не можете рассчитывать на то, что всю свою жизнь только и будете делать, что поражать своих восхищенных профессоров виртуозной техникой программирования. Короче, быть художником, одиноким и стоящим от всего в стороне, вам не удастся; маловероятно, чтобы из среды программистов когда-либо вышел Микельанджело или Рембрандт. Как программист вы всегда должны будете делать все необходимое, чтобы заставить вычислительную машину выполнять определенные полезные операции. И это так независимо от того, программируете ли вы экономические или научные задачи или участвуете в исследованиях по разработке вычислительной машины.

Таким образом, я считаю, что обсуждение принципиальных вопросов программирования является делом в высшей степени практическим. Что вам необходимо сделать, чтобы стать хорошим программистом? Какие свойства ваших программ наиболее важны для вашего нанимателя? Эти вопросы, возможно, покажутся вам пустячными, но от их решения может зависеть ваш успех как программиста. Пренебрежение принципами, рассмотренными в этой главе, обойдется вам дорого, если иметь в виду ваше продвижение по службе. Это может даже стоить вам места. Что же может быть еще ближе к практике?

Основное назначение этой главы — определить список контрольных вопросов, позволяющих выявить особенности хорошей программы. Мы начнем с общих замечаний по поводу того, какие качества характеризуют хорошего *программиста,* а также — хорошую *программу;* затем более конкретно будут рассмотрены семь характерных особенностей хорошей программы. Написав программу для ЭВМ, вы должны спросить себя, отвечает ли ваша программа этим «правилам» хорошего программирования; *перед* написанием программы вы должны уяснить себе, с помощью каких средств вы можете наилучшим образом учесть эти правила.

1.1. Какими качествами обладает хороший программист?

В течение нескольких последних лет я имел возможность читать курс программирования повышенного типа тысячам слушателей в разных странах мира. Как правило, это были опытные программисты, работающие в банках, страховых компаниях,» правительственных учреждениях, производственных и научных организациях, университетах и прочих мыслимых учреждениях. Больше для собственного развлечения, чем для какой-нибудь иной цели, я

часто начинал чтение курса следующим вопросом: «Какими качествами должен обладать хороший программист?» Ответы бывали столь же различными, как и профессиональное прошлое самих слушателей, и некоторые из них стоит процитировать:

1. Хороший программист пишет хорошие программы (или эффективные программы, или хорошо оформленные программы и т. д.).
2. Хороший программист умеет работать с другими людьми.
3. Хороший программист умеет общаться с пользователями его программы.
4. Хороший программист моется по крайней мере раз в неделю.
5. Хороший программист приходит на работу вовремя.
6. Хороший программист *никогда* не приходит на работу вовремя.
7. Хороший программист не доставляет хлопот.
8. Хороший программист умеет работать в напряженной обстановке.
9. Хороший программист любит классическую музыку.

Часто возникают споры по поводу первого ответанаэтотвопрос, т. е. утверждения, что хороший программист — тот, кто пишет хорошие программы. Известно, что оценка «качества» программиста входит в обязанности руководителей, но они, кажется, менее, чем кто-либо другой, в состоянии оценить качество хорошей программы. Из содержания этой главы станет ясно, что в действительности *никто* из нас не чувствует себя уверенно в количественных оценках хороших программ, так что в этой трудной ситуации не следует преувеличивать вину руководителя.

Вопреки сказанному кажется, что некоторые программисты пользуются в своих организациях репутацией «суперпрограммистов». В этом случае говорят, например, что Том может написать большую сложную программу за один день или что Элис всегда удается отладить программу максимум за один контрольный пуск. Исходя из такой естественной ситуации, я иногда изменял первоначальный вопрос и спрашивал своих слушателей-программистов:

> Пользуется ли суперпрограммист — т. е. тот, кто прямотаки стреляет программами, кто одним взглядом может оценить содержание целой стопки бумаг с распечатками и результатами, кто обойдет любого в своей организации во всем, что касается программирования,— особым расположением и уважением руководства?

Хотя по этому вопросу часто возникают горячие споры и высказываются противоположные мнения, удивительно много людей, особенно из числа руководителей групп программирования, на этот вопрос отвечают решительным «Нет!». Программист одного крупного банка в Монреале выразил это довольно точно: «Если мои программы ужасающе неэффективны, настолько неэффективны, что это понятно даже моему руководителю, то у меня могут быть неприятности. Если я пишу программу в десять раз дольше, чем другие со-

трудники отдела, то это также чревато неприятностями. Однако больше всего моего руководителя интересует моя способность функционировать в качестве одногоиз членов определенного коллектива людей; мои отношения с вычислительной машиной остаются моим личным делом. Для руководителя важно, чтобы я работал в достаточно определенные часы, хорошо взаимодействовал с другими программистами и пользователями ЭВМ, асамоеглавное — не доставлял хлопот». Представляется, что такое отношение руководства преобладает в крупных страховых компаниях, правительственных учреждениях и банках, менее свойственно производственным предприятиям средних размеров и обычно не характерно для университетов, научно-исследовательских организаций и компаний, разрабатывающих вычислительную технику.

Если руководство занимает подобную позицию, то вопросы совершенствования программирования, как и содержание этой книги, представляют скорее академический интерес. Какой смысл в написании самых эффективных в мире программ, когда вы лишены тех качеств, которые необходимы, чтобы пользоваться признанием, если не быть любимцем, в вашей организации? К счастью, положение обычно не столь безнадежно. В действительности руководству хотелось бы знать, что используемые программы обладают всеми «хорошими» свойствами, которые мы будем обсуждать в этой главе; важно только помнить, что руководители редко терпят известную атмосферу «компьютерного бума», которую оправдывают разработкой высокоэффективных программ.

Заметим, что существуют различные причины, которыми объясняется антипатия к так называемым суперпрограммистам. Некоторые суперпрограммисты могут делать программы очень быстро или писать очень эффективные программы, но эти программы иногда остаются неоформленными, их невозможно понять, сопровождать или модифицировать. Или другое: встречаются суперпрограммисты, которые создают выдающиеся программы, но эти люди трудны в общении, или, говоря словами одного руководителя, «что-то вроде Аллена Гинзберга» [1)].

Самое примечательное в суждениях о качествах хороших и плохих программ состоит в том, что мы, оказывается, не умеем их оценивать посредством какой-нибудь разумной количественной меры, т. е. мы представляем, что некий суперпрограммист пишет программы, которые намного лучше программы его рядовых коллег, но сказать, насколько лучше, мы не можем. В этой области мы не чувствуем себя уверенно, у нас нет способа установить, что он, скажем, вдвое или в 3,14, или в 10 раз лучше других программистов.

[1)] Американский поэт-авангардист, известен своей эксцентричностью,— *Прим. перев.*

Этот факт был очень выразительно проиллюстрирован в исследованиях, проведенных Сакманом, Эриксоном и Грантом[1]. В ходе исследований по анализу преимуществ работы в режиме разделения времени в сравнении с режимом пакетной обработки[2] двенадцати опытным программистам было предложено составить программы для решениядвух задач:одна задача была алгебраической.другая — логической, отыскивающей выход из лабиринта («крыса в лабиринте»). Для определения времени, затраченного на отладку и написание программы, велись тщательные записи. Считалось, что отладка начинается с того момента, как программист выявил все «серьезные» синтаксические ошибки, и заканчивается, как только программа получает правильное решение стандартной контрольной задачи.

Результаты этого эксперимента представлены в табл. 1.1. Обратите внимание на поразительные различия в лучших и худших

Таблица 1.1

ДИАПАЗОН ИНДИВИДУАЛЬНЫХ РАЗЛИЧИЙ В ОПЫТЕ ПРОГРАММИРОВАНИЯ

Измеряемая характеристика	Худшее значение	Лучшее значение	Отношение худш./лучш.
1. Отладка алгебраической программы, ч	170	6	28:1
2. Отладка логической программы, ч	26	1	26:1
3. Время ЦП 1) на разработку алгебраической программы, с	3075	370	8:1
4. Время ЦП на разработку логической программы, с	541	50	11:1
5. Написание алгебраической программы, ч	111	7	16:1
6. Написание логической программы, ч	50	2	25:1
7. Объем алгебраической программы	6137	1060	6:1
8. Объем логической программы	3287	650	5:1
9. Время ЦП решения алгебраической задачи, с	7,9	1,6	5:1
10. Время ЦП решения логической задачи, с	8,0	0,6	13:1

1) ЦП — центральный процессор.

[1] H. Sackman, W. J. Erickson, E. E. Grant, Exploratory Experimental Stadies Comparing On-line and Off-line Programming Performance, Communications of the ACM, Jan. 1968, p. 3—14.

[2] Интересно отметить, что, как установили исследователи, режим разделения времени действительно способствует более быстрому завершению работы, хотя при этом на разработку программ программисты расходуют больше машинного времени.

значениях характеристик, касающихся написания и отладки программ. Соответствующие отношения для остальных параметров, хотя и не столь выразительны, тоже достаточно велики, так что руководителю не просто спланировать и составить график работпо созданию программы. Как выражаются авторы в своих выводах:

Если программист хорош,
То он очень и очень хорош,
Но уж если он плох,
То просто ужасен.

Авторы также отмечают, что «характер зависимостей, выявленных в эксперименте, хорошо согласуется с оценками обучающихся на курсах программирования, однако не обнаруживается какой-либо корреляции, характеризующей опытных программистов». В окончательных выводах авторы утверждают следующее.

«Полученные результаты говорят о том, что общие навыки программирования могут быть важнее, чем предварительное обучение или освоение начал программирования «в рабочем порядке». При этом общие навыки по мере накопления опыта постоянно меняются, все более и более превращаясь в особое профессиональное мастерство».

Какие выводы можно сделать из всего сказанного? Не многие, за исключением того, что мы очень мало знаем о том, что делает программистов плохими или хорошими, или о том, как определить объективную меру их способностей. Кажущаяся тавтология «хороший программист — тот, кто пишет хорошие программы», может оказаться и ошибочной, если учесть мнение тех, кто определяет повышения и продвижениепо службе.Помимо знания ОС[1] и JCL[2] Системы IBM/360, хороший программист, оказывается, должен обладать рядом качеств, не имеющих никакого отношения к вычислительным машинам.

1.2. Какими качествами должна обладать хорошая программа?

Как я упоминал ранее, я часто начинаю чтение курса программирования повышенного типа, предлагая слушателям сформулировать определение хорошего программиста. Не удивительно, что за этим предложением часто следует вопрос: «Какими качествами обладает хорошая программа для ЭВМ?». И на этот раз предлагается множество определений; ниже приводятся некоторые из наиболее интересных:

[1] ОС — операционная система.— *Прим. перев.*
[2] JCL — язык управления заданиями.— *Прим. перев.*

1. Она работает.
2. Она работает согласно спецификациям.
3. Она гибкая.
4. Она сделана в срок.
5. В ней нет ошибок.
6. Ошибки, которые неизбежны, быстро могут быть выявлены.
7. Она хорошо оформлена.
8. Она решает задачи быстро.
9. В ней эффективно используется память.

Очевидно, что мы могли бы добавить сюда еще ряд желательных качеств программы, но и приведенный список выглядит достаточно представительным. Следовало бы, однако, упорядочить и стандартизировать эти желательные качества с тем, чтобы каждый мог воспользоваться некоторым общим критерием, позволяющим оценивать написанные им программы. Ниже дается перечень семи желательных качеств программ, перечисленных, как я считаю, в порядке убывания степени их важности. Обсуждая эти качества, мы будем также комментировать наши возможности их *количественной* оценки.

1.2.1. Программа работает и легко анализируется

Интересно отметить, что многие их моих слушателей уже на первом-втором году программирования устанавливают для себя, что *самое важное свойство программы заключается в том, что она работает*. И просто невероятно, что так часто приходится наблюдать, как два программиста (или две организации, разрабатывающие математическое обеспечение, или две фирмы, выпускающие вычислительные системы) вступают в спор о сходных по назначению программах, который звучит так:

Программист А: «Моя программа в десять раз быстрее вашей, и она занимает в три раза меньше памяти!»

Программист Б: «Да, но ваша программа не работает, а моя — работает!»[1]».

В некоторых случаях имеет смысл делать различие между программой, которая работает, и программой, которая работает согласно техническому заданию. По ряду понятных причин программист в итоге может учесть лишь часть исходных требований или, может быть, составить программу, решающую совершенно иную задачу: программист может ошибочно истолковать требования технического задания, сами требования могут оказаться неоднозначными или неточными (хорошим средством исключения этого положения могут служить таблицы решений), требования могут изме-

[1] Весьма занятную историю в этой связи, а также многие другие тонкие наблюдения из области психологии программирования можно найти в книге G. Weinberg, The Psychology oi Computer Programming, D. Van Nostrand Co., 1971.

ниться за время разработки программы или программист может вдруг обнаружить, что наобещал больше, чем может сделать, и реализует лишь частьисходных требований.

Программы и системы программирования неизменно усложняются и поэтому в будущем нам, вероятно, будет недостаточно того, что программа просто работает; могут потребоваться какие-то методы *верификации* правильности работы программы (а может быть, и программиста). Обратите внимание на тонкое различие между *тестированием* программы и проведением некоторой *систематической процедуры проверки {верификации)* правильности программы. Исход тестирования программы важен прежде всего для программиста и того, кому приходится принимать законченную программу (т. е. руководителя программиста или представителя пользователя). Проверка программы укрепляет в человеке, который должен использовать полученные с ее помощью результаты, чувство уверенности в том, что программа действительно работает правильно.

Проверка такого рода может быть необязательной во многих простых прикладных задачах, но она была бы полезна в тех случаях, когда вычисления столь сложны, что не поддаются непосредственной оценке пользователем. Рассмотрим, например, какую-нибудь инженерную задачу, в которой пользователю необходимо провести расчеты, облегчающие выбор параметров и размеров моста, самолета или какого-нибудь другого дорогого и сложного сооружения. Вычисления, выполняемые программой, могут включать решение тысяч дифференциальных уравнений, численное интегрирование, решение систем уравнений и т. д., и тем не менее окончательным результатом может быть такое простое утверждение, как «крылья самолета должны иметь размах 972,34567 фут». У инженера этот результат может вызвать большое подозрение, так как ему никогда не приходилось видеть самолет с такими огромными крыльями, и тем не менее ответ может быть правильным. В таком случае было бы полезно, если бы в программе предусматривалась возможность пояснять, *каким образом* она пришла к такомуответу; этоможно было бы сделать выдачей промежуточных результатов вычислений.

Повторяем, что такой анализ не обязателен во многих простых задачах; однако в сложных экономических и научно-иеследовательских расчетах, где способы проверки, доступные пользователю, оказываются недостаточными, эту особенность программ для ЭВМ следует рассматривать как важнейшую.

1.2.2. Минимальные затраты на испытания

Тестирование и отладка программы для ЭВМ, несомненно, составляют главный предмет беспокойства руководителя программистов; часто это главная забота и для программистов. В среднем в области производства вычислительных машин тестирование и отладка прог-

рамм занимают от 1/3 до 1/2 полного времени разработки. Ясно, что любые способы снизить эти чрезмерные потери очень важны.

Принципиальная точка зрения такова, и это нужно помнить, что если вы написали столь умную, изощренную и сложную программу, что она понятна только вам, то эта программа бесполезна; если в вашей программе используются скрытые, известные только посвященным возможности машины, то ваша программа бесполезна; если в вашей программе отсутствуют комментарии и она не оформлена, то программа также бесполезна. Увлечение такими качествами приводит к созданию бесполезных программ, поскольку их проверка *весьма* затруднительна. В итоге программы оказываются недопустимо дорогими в смысле затрат средств и времени.

Говоря о затратах средств и времени, заметим, что эти характеристики программы могут быть оценены количественно; мы можем измерить качество программы с точки зрения затрат на ее испытания, фиксируя время, затраченное на испытания, и время центрального процессора, использованное на контрольные пуски, учитывая специальные вспомогательные меры на время испытаний (например, сложные аппаратные имитаторы систем реального времени или дорогостоящие режимы параллельной работы) и потери из-за утраченного доверия заказчика в связи с непрошедшими испытания программами. Как показывает табл. 1.1, количество времени, расходуемое программистами на испытания, может изменяться в очень широком диапазоне.

Из практики известен ряд часто используемых порочных приемов, которые вызывают трудности при отладке программ; ниже рассматриваются некоторые из худших примеров.

Программа без комментариев. На мой взгляд, в области программирования ничто не заслуживает большего порицания, чем программа без комментариев. Программисту можно простить многие прегрешения и фантазии, не исключая и тех, что перечислены ниже, однако *ни одному* программисту, как бы ни был он умен и опытен, как бы ни было у него мало времени, как бы ни хороши были его намерения, нельзя простить программу, не оформленную и не содержащую комментариев.

Если эта критика покажется вам не в меру суровой, то попробуйте отладить, обслуживать (сопровождать) или модифицировать чужую программу без комментариев; скорее всего вы убедитесь, что это хуже, чем не иметь никакой программы. Подробный комментарий является хорошей предпосылкой к быстрой и легкой отладке ваших собственных программ. Только глупец может решиться идти незнакомым лесом, не оставляя за собой знаков. Писать программы без комментариев — это то же, что пробираться джунглями Амазонки с завязанными глазами. В этом отношении нет твердо установленных правил, но можно руководствоваться следующим: от четырех до пяти строк комментария на каждую подпрограмму (или на сек-

цию в Коболе и т. д.) и в среднем по одному комментарию на каждые две-три строки исходной программы.

Важно подчеркнуть, конечно, что комментарии не являются самоцелью. Как заметили Керниган и Плогер в их замечательной книге The Elements of Programming Style, хороший комментарий не заменит плохих кодов. Однако совсем не очевидно, что хорошая кодовая часть программы может служить и комментарием. То есть я не согласен с тем, что «хорошие» коды не нуждаются в комментарии: коды говорят нам о том, что делается программой, комментарий же часто необходим, чтобы понять, *почему* программист использует те или иные операторы.

На своем опыте программиста и ответственного руководителя программистов я имел возможность слышать (а порою и использовать) следующие доводы (на самом деле *оправдания)* в связи с отсутствием комментариев в программах:

1. У меня нет достаточно времени, чтобы делать какие-либо комментарии.
2. Мне приходится самому перфорировать программы, а печатаю я плохо, поэтому не могу тратить много времени на подробный комментарий.
3. Я составляю свои программы с терминала в режиме разделения времени и отвечаю за использование времени связи, а поскольку я печатаю плохо, я не могу включать в программу подробный комментарий.
4. Моя программа ясна сама по себе.
5. Всякий *компетентный* программист может понять мою программу без пояснений.
6. Моя программа будет использована только один раз, так что в ее оформлении нет никакой необходимости.
7. Эта программа наверняка подвергнется серьезным изменениям во время отладки и испытаний и к моменту их завершения комментарий уже устареет.
8. Я прекрасно понимаю, что делает моя программа,— зачем мне нужно ее комментировать?
9. Я не люблю оформлять или комментировать.
10. Иметь слишком много комментариев вредно — это делает неясным смысл наиболее важных из них.
11. Если я включу много комментариев, то компиляция моей программы займет много времени.
12. Моя исходная программа будет занимать слишком много места на диске (или на перфоленте в миникомпьютерных системах), если я буду писать много комментариев.
13. Пояснения, да кто их вообще читает?

Ясно, что отсутствие комментариев чревато трудностями; однако еще большие трудности могут возникнуть в случае, когда программа насыщена комментариями, но при этом:

1. Комментарии избыточны.
2. Комментарии устарели.
3. Комментарии были и остаются ошибочными.
4. Комментарии неоднозначны и неточны.
5. Комментарии правильны, но неполны.
6. Комментарии не понятны никому, кроме автора.

Именно это свойство комментариев представляет потенциальную опасность. Комментарий, содержащий ложную или вводящую в заблуждение информацию об операторе программы, которому он сопоставлен, может быть вреднее, чем отсутствие всякого комментария; избыточный комментарий (например, комментарий «теперь мы пересылаем А в В», сопровождающий в Коболе оператор MOVE A TO B) может настолько обескуражить программиста, обеспечивающего сопровождение программы, что он вообще перестанет искать полезные комментарии в программе. Не удивительно поэтому, что, согласно некоторым экспериментам, чужую программу легче и быстрее отладить, если сначала из нее изъять все комментарии.

К сожалению, у многих программистов комментарий выглядит как «письма себе», т. е. личные заметки для памяти, поясняющие использование тех или иных конкретных операторов или команд в их программе. Смысл таких личных заметок, однако, может оказаться понятным только их автору, и даже автор спустя некоторое время затрудняется в понимании таких комментариев (а следовательно, и операторов, которые они поясняют). Несколько лет назад произошел следующий занятный случай такого рода. Суперпрограммист-одиночка разработал компилятор для Фортрана-П по заказу известной организации, производящей ЭВМ. Проведя испытание компилятора, программист сдал работу руководителю и на несколько дней исчез[11]. За это время руководитель установил, что в про-

[11] Позже выяснилось, что наш суперпрограммист прочел в местной газете, что некий старшекурсник из Гарварда проехал за один прием все линии нью-йоркского метро (включая все ветки линий IRT, IND и BMT) в рекордное время (так как ранее никто не пытался этого сделать), равное 48 ч. Будучисамвыпускником МТИ, суперпрограммист счел это вызовом в здоровом межинститутском соревновании; он составил программу, которая должна была вычислить наискорейший путь, включающий все линии метро, и подобрал команду друзей с тем, чтобы взяться за решение проблемы с высоконаучных позиций. Программа, написанная на Лиспе и включающая весьма сложные эвристические алгоритмы решения этого варианта задачи коммивояжера, израсходовала ресурсы памяти вечером как раз накануне того дня, когда команда должна была выехать в Нью-Йорк и ринуться в подземку. Оказавшись перед выбором между оптимизацией пути, включающего отдельную линию или принятием «субоптимальной» стратегии обхода всех линий, команда избрала путь частичной оптимизации всей транспортной системы. Установить, работала ли эта программа правильно, так и не удалось, поскольку после 27 ч в подземке один из членов команды в критический момент заснул на одной из станций; пока остальные члены команды возвращались и разыскивали потерянного товарища, нужный поезд, связывающий линии, ушел без них — в итоге они проиграли студенту из Гарварда около 5 мин. Стоит ли говорить о том, что все это не показалось забавным руководителю отдела программирования?

грамме имеется ряд ошиоок, треоующих немедленного исправления. Младший программист, которому было поручено проанализировать ошибки, к своему ужасу обнаружил, что вся программа компилятора содержит единственный комментарий к восьмеричной константе следующего вида:

```
CONST23:  3443  ; R.I.P.L.V.B.
```

Поскольку, по отзывам, суперпрограммист отличался лаконичностью и блестящими способностями, младший программист решил, что, возможно, этот единственный комментарий послужит ключом ко всем тайнам компилятора. После нескольких часов размышлений над смыслом комментария он, наконец, нашел ответ — восьмиричное целое 3443 эквивалентно десятичному 1827. Будучи любителем классической музыки (вспомните приведенный ранее список особенностей хорошего программиста) и собирателем тривиальной информации, младший программист к случаю вспомнил, что в 1827 г. умер Бетховен! Можно себе представить, как мало забавного нашел во всем этом руководитель отдела программирования, и когда суперпрограммист появился вновь, ему было предложено, чтобы он демонстрировал свои неоценимые таланты где-нибудь в другом месте.

Использование языка ассемблера вместо языков высоких уровней. Программирование на языке ассемблера — своего рода фетишизм в кругу «искушенных» программистов; эта болезнь особенно распространена среди студентов старших курсов вычислительных специальностей. Замечено, что в профессиональной градации программистов, существующей во многих организациях, программисты, работающие на языке ассемблера, считаются наиболее квалифицированными.

Следует, однако, помнить, что программы, написанные на языке ассемблера, проверять почти всегда значительно труднее, чем программы на языках высокого уровня; для неосмотрительного программиста использование языка ассемблера чревато многими скрытыми осложнениями. В большинстве случаев экономических и научных применений языки высокого уровня оказываются вполне приемлемыми: более того, растет уверенность в том, что даже *системные программы,* например компиляторы, ассемблеры, операционные системы и т. д., следует писать на языке высокого уровня.

В тех случаях, когда необходима крайне высокая эффективность программ, использование языка ассемблера может оказаться оправданным. Однако более разумным представляется другой подход, когда первоначальная версия программы пишется на Фортране, Коболе или ПЛ/1, затем определяются те фрагменты программы, которые в наибольшей степени требуют оптимизации, и только эти фрагменты переписываются на языке ассемблера.

Многоцелевые и асинхронно используемые программы. Многоцелевое использование — одно из нескольких часто употребляемых

выражений, применяемое для описания методов, которые позволяют программисту инициировать и проводить решение нескольких различных задач с помощью одной и той же программы. Это очень полезный и важный подход, особенно в области получающих все большее распространение диалоговых систем и систем реального времени.

Именно потому, что этот подход придает прикладной программе некоторые черты, обычно относимые к функциям только операционных систем (т. е. обеспечение одновременной работы нескольких программ), отладка таких программ становится намного труднее. Если вы разрабатываете не диалоговую систему или систему реального времени, где фактор эффективности имеет огромное значение, то этого подхода следует избегать; в других случаях, например при обеспечении параллельного выполнения операций ввода-вывода и вычислений в системах пакетной обработки программ, прежде чем воспользоваться этим подходом, вам следует убедиться в том, что выигрыш в машинном времени компенсирует дополнительные 10—15% затрат времени на испытания.

Файлы общего пользования в мультипрограммных режимах. Еще одной причиной трудностей мультипрограммирования является использование общих файлов. Фактически все вычислительные системы третьего поколения допускают мультипрограммный режим; они, по сути дела, на него рассчитаны. Во многих системах допускается также, при соблюдении некоторых разумных условий защиты, использование различными одновременно выполняемыми прикладными программами одних и тех же файлов.

И в этом случае необходимо отдавать себе отчет в том, что этот мощный метод таит в себе определенную опасность. Некоторые системы, например, допускают одновременное считывание файла одной программой и обновление его другой. На рис. 1.1 показана сложная ситуация, которая может возникнуть: если одна программа считывает две или больше логически связанных записей в то время, когда другая программа их обновляет, то первая программа может считать несогласованную информацию.

Нестандартное использование операций. Склонный к ухищрениям программист обычно может найти некоторый способ нестандартного, непредусмотренного применения любой машины, любого языка программирования или операционной системы. Он отыщет такие возможности, которые, хотя и не описаны ни в какой документации разработчика ЭВМ и математического обеспечения, окажутся применимыми в его задаче и, возможно, сэкономят несколько микросекунд машинного времени или несколько ячеек памяти. В машинах второго поколения, например, программисты могли иногда пользоваться так называемыми неопределенными машинными командами, т. е. кодами операций, не описанных в руководствах по языку ассемблера, которые приводили к довольно странным и не пред-

сказуемым результатам. ЭВМ IBM 7090/7094 явилась классическим примером машины, которая может быть использована программистом таким непредусмотренным образом.

Хотя в большинстве машин третьего поколения исключена возможность употребления неопределенных операций, программист все-таки может нестандартно использовать допустимые операции. В некоторых машинах,например,программисты пользуютсяоперациями с нормализацией и плавающей точкой для выделения значения первого бита поразрядной маски; другие находят какое-то неожиданное применение неиспользованных разрядов машинного «слова состояния программы» (PSW); третьи могут воспользоваться тем фактом, что случайно программа устанавливает некоторый режим работы машины при первом проходе цикла и не делает этого при повторных проходах.

Конфликтная ситуация начинается с того, что программа *A* считывает запись *X,* загем она будет считывать запись *Y.* В это время программа *B* приступила к обновлению записи *Y,* затем она будет обновлять запись *X.*

В этот момент программа *A* считала запись *Y* и этопривело к ошибке: программой *A* считана „старая" запись *X* и „новая" запись *Y.*

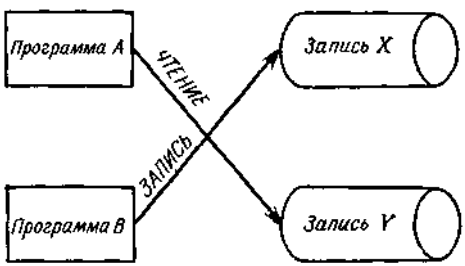


Рис. 1.1. Конфликтное использование общих файлов в мультипрограммном режиме.

Программисты на языках высокого уровня также склонны к этим слабостям. Рассмотрим, например, следующий фрагмент программы на Фортране:

```
      DIMENSION A (20), B (30)


      DO10I=1,50
10    A(I)=J
```

В этом примере программист учел тот факт, чтопослекомпиляции Фортран-программы и загрузки ее в память два упомянутых массива оказываются расположенными один под другим. В результате циклом по оператору DO будет задан весь массив А и вслед за ним — массив В, при этом не нужно писать заголовок второго цикла, формирующего начальное состояние массива В. Хотя такой прием выполним на некоторых машинах (я сам впервые обнаружил это в Фортране CDC 6400,работая над этой книгой) ихотяэтодействительно экономит время центрального процессора и некоторую память, его следует считать нарушением правил языка Фортран.

В Фортране возможны и некоторые нестандартные употребления. В более ранних версиях Фортрана студентами-программистами в нескольких университетах было обнаружено, что следующая последовательность операторов приводит к довольно интересным результатам:

```
      ASSIGN  10  TO  I


      I=12345


      GO TO I
```

Если Фортран-компилятор не фиксирует это как ошибку (большинство современных компиляторов это делает), то оператором GO TO управление будет передано ячейке памяти 12345. Такой прием часто оказывается весьма удобным при формировании альтернативных входов в подпрограмму.

Известно, что программисты на Коболе также прибегают к непредусмотренным применениям этого языка. Один из наиболее распространенных тому примеров — использование поля записи в качестве рабочей памяти после того, как файл был открыт, но прежде, чем выполнены операторы ввода-вывода, или, наоборот,

```
        ALTER SWITCH1 TO PROCEED TO SWITCH2.

SWITCH1
        GO TO INITIALIZATION-ROUTINE

SWITCH2
        ALTER SWITCH3 TO PROCEED TO SWITCH4.
```

Примечание. Если эти инструкции хаотично распределены в большой программе, то бывает очень трудно понять, что такая программа делает. В дополнение программа может стать *нереентерабельной* или исключающей повторное использование; мы вернемся к обсуждению такой возможности в гл. 5.

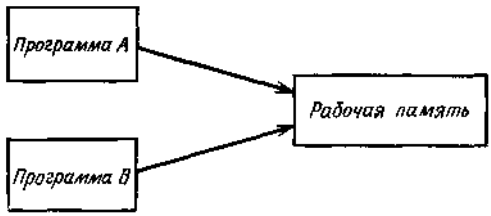
Рис. 1.2. Модифицирующаяся Кобол-программа.

после того, как закончены операции обмена, но файл еще не закрыт. Программисты, работающие на Коболе и Фортране, склонны предполагать, что до начала работы их программ все значения переменных, имен данных, таблиц и массивов нулевые.

Все это не только не полезно, но и опасно. Программист только тратит лишнее время, принуждая машину (или компилятор, или операционную систему) исполнять его нестандартные команды, рискуя при этом разрушить логику программы.

Программы, которые себя модифицируют. Порочная практика этого приема в программировании будет неоднократно обсуждаться в последующих главах. На рис. 1.2 показан пример модифицирующей себя программы; некоторые программисты доводят реализацию этого принципадо невероятных масштабов. Например, команда А (в начале программы) может, при некоторых неясных заранее условиях, изменить команду В (в середине программы), которая в свою очередь изменяет команду С (в конце программы). Пристрастившийся к этому приему программист еще устроит так, что команда С модифицирует команду А. Результирующая программа обнаруживает замечательное сходство с хамелеоном, и ее почти невозможно отладить.

Общие переменные и временная память нескольких подпрограмм. Многие языки высокого уровня позволяют программисту объявлять общие участки временной памяти, и эта идея может быть полезной при работе с большими массивами, зонами промежуточной памяти и т. д. Она часто используется и в программах на языке ас-



Программа *A* и программа *B* могут использовать общую рабочую память, когда они логически независимы; т. е. если *A* не вызывает *B* и *B* не вызывает *A,* то конфликта быть не может.

Рис. 1.3. Разделение рабочей памяти между программами.

семблера: две подпрограммы часто располагают результаты промежуточных вычислений в одних и тех же ячейках памяти. Во многих случаях этот обычай введения общей памяти оказывается несколько сложнее, чем кажется. В Фортране, Коболе и ПЛ/1, например, программисты часто используют в одном и том же модуле одни и те же переменные I, J, K для совершенно различных целей.

Как показано на рис. 1.3, практика использования общей памяти допустима лишь в том случае, когда модули оказываются независимыми; иными словами, если одна подпрограмма не вызывает другую, то трудностей не возникает. Однако логические взаимосвязи подпрограмм (или разделов, или секций в Кобол-программах) могут изменяться в процессе создания и испытания всей программы. Таким образом, весьма вероятно, что программы, которые прежде совместно работали, вдруг оказываются в противоречии; ошибки такого рода часто бывает *крайне трудно* выявить!

Даже в том случае, когда программы являются логически независимыми, они могут оказаться в конфликте, употребляя общую временную память. Как это иллюстрирует рис. 1.4, две подпрограммы, выполняемые в мультипрограммном режиме, могут прерывать друг друга и портить содержимое ячеек временной памяти. Поскольку эти прерывания связанны с временными характеристиками работы операционной системы, выявление этих ошибок оказывается еще более сложным (например, причиной в некоторых случаях может оказаться то, что записи одной программы считываются с диска *чуть-чуть* раньше записей другой, или то, что одна подпрограмма получает управление вводом с терминала на миллисекунду раньше, чем другая).

Чтобы избежать этих трудностей, нужно пользоваться следующим хорошим правилом: отводить каждому модулю или подпрограмме отдельные зоны временной памяти. Это достигается естественным образом в языках типа ПЛ/1 или Алгол, но требует некоторых сознательных усилий в Фортране, Коболе и языке ассемблера. Введением общей памяти для разных подпрограмм следует поль-

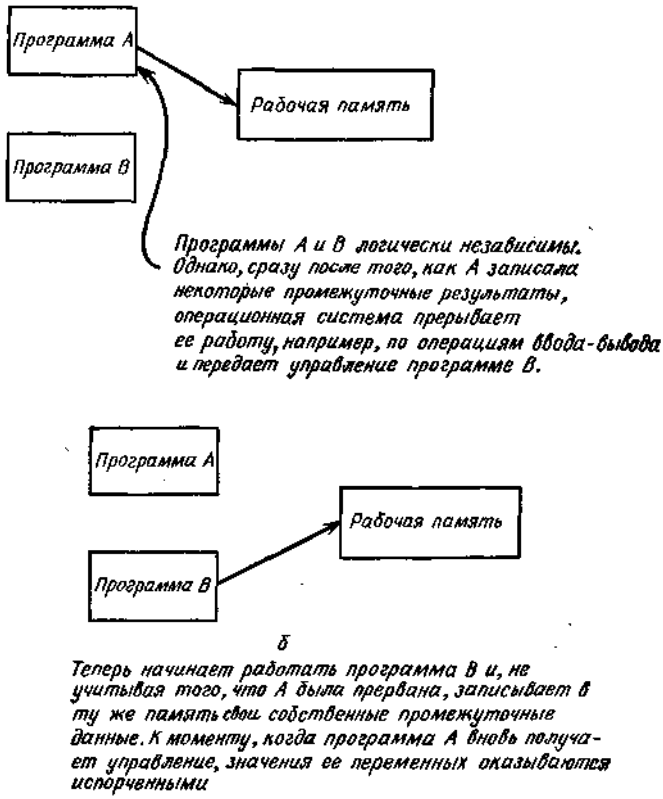


Рис. 1.4. Конфликтная ситуация пользования общей рабочей памятью в системе реального времени.

зоваться только в тех случаях, когда этим достигается значительная экономия памяти (например, несколько сотен ячеек одним массивом).

Сложные макрокоманды на языке ассемблера. Большинство средних и больших ЭВМ оснащено довольно развитой системой макрокоманд в языке ассемблера. Если вы не знакомы с макрокомандами, пожалуйста, забудьте на некоторое время об этом разделе; в данном случае незнание есть благо. Если вы уже знакомы с макрокомандами, то вам следует ими пользоваться ограниченно и просто. Избегайте вложенных и рекурсивных макрокоманд, а также макрокоманд, которые сами себя переопределяют; все это интересно для специальных курсов в теории программирования, но на практике имеет очень ограниченное применение. «

Если вы *вынуждены* применить макрокоманду (например, для формирования очень сложных таблиц или организации очередей),

то обязательно вносите несколько комментариев в окрестности определения вашей макрокоманды. Следует также тщательно проверять текст программы, формируемой макрооператором; если вы определите очень сложную макрокоманду, то может случиться, что ассемблер не в состоянии ее реализовать.

Программы, перегруженные операторами GO TO. Как в языках высокого уровня (Фортран, Кобол, ПЛ/1 и т. д.), так и в языке ассемблера программы с чрезмерным употреблением оператора GO TO, т. е. безусловной передачи управления, оказываются весьма трудными в отладке. Таким программам свойственно отсутствие какой-либо организации или структуры управления. Вместо того чтобы строить программу на основе управляющих операторов GO TO, программисту следует прибегнуть к методам организации, основанным на таблицах решений, подпрограммах или блочных структурах BEGIN — END, известных в ПЛ/1 и Алголе.

В табл. 1.2 приводится некоторая статистика по программе на Фортране, в которой, по-видимому, использовано чрезмерное количество операторов GO TO. Заметьте, что среднее число операторов GO TO по всей программе поразительно велико—24,2%, но оно еще выше в некоторых крупных подпрограммах и в главной программе. Дело усугубляется еще тем, что некоторые операторы GO TO употребляются хаотично, управление передается то вперед, то назад на несколько страниц листинга программы, и поэтому следовать логике программы крайне трудно. Кодирование этой программы заняло необычайно много времени, и она оказалась очень трудной в отладке — до сих пор она еще не работает должным образом[1)]. Для сравнения интересно отметить, что, как установил Кнут[2)], среднее значение операторов GO TO в Фортран-программах составляет 13%.

В некоторых случаях излишнее употребление операторов GO TO объясняется небрежным программированием. Например, часто можно увидеть такую последовательность кодов в Фортране:

```
      IF (X .EQ. 0) GO TO 20
      A=17
10    CONTINUE


      A = 23
20    GO TO  10
```

[1)] Более подробный анализ этой очень интересной программы можно найти в книге E. Yourdon, Measuring the Goodness of a Computer Program, ACPA Thruput, April 1972.

[2)] D. E. Knuth, An Empirical Study of FORTRAN Programs,Software-Practice and Experience, **v. 1,** № 2, April-June **1971, p. 105—133.**

Таблица 1.2

АНАЛИЗ ПРИМЕРА ПЛОХОЙ ФОРТРАН-ПРОГРАММЫ

	Главная программа	Подпрограммы										Всего
		HDNG5	MESSET	MSFLD	CHECK	DSTCT	STTRPT	ANA	HDHG6	HGNG7	TWEEK	
Всего карт в программе	1354	54	346	78	141	152	128	170	39	49	67	2578
Всего карт комментария	626	27	159	45	61	60	58	74	16	19	38	1183
Карт с пустым комментарием	351	8	70	24	§3	28	36	47	8	9	17	651
Карт с непустым комментарием	275	19	89	21	28	32	22	27	8	10	21	532
Карт с операторами Фортрана	728	27	187	33	80	88	70	96	23	30	29	1395
Карт с операторами описаний	204	14	14	5	24	25	17	27	14	15	12	371
Карт с исполняемыми операторами	525	13	173	28	56	63	53	69	9	15	17	1024
Операторов ввода-вывода	25	6	0	0	14	11	5	15	5	6	3	90
Операторов DO	27	0	26	5	2	1	3	5	0	0	2	71
Операторов IF	67	3	3	4	12	7	7	8	1	2	3	117
Операторов GO TO	141	2	49	8	24	10	10	11	1	3	4	263
Операторов присваивания (RETURN и т. д.)	251	4	90	13	12	38	28	32	2	4	4	478
Операторов передачи управления	13	1	**3**	1	2	5	3	2	1	1	4	36
Операторов CALL	14	0	**0**	0	0	2	4	5	0	0	0	25
Вычисляемых операторов GO TO	2	1	**1**	0	0	0	0	0	0	0	0	4
Всего исполняемых операторов	540	17	172	31	66	74	60	78	10	16	20	1084

Интересные статистические данные:

46% всего количества карт программы составили карты комментариев
55% карт комментариев — пустые
26,6% карт, не являющихся картами комментариев, заняты операторами описаний
73,4% карт, не являющихся картами комментариев, заняты исполняемыми операторами
8,3% исполняемых операторов составили операторы ввода-вывода
6,5% исполняемых операторов составили операторы DO
10,8% исполняемых операторав составили операторы IF
24,4% исполняемых операторов составили операторы GO TO
44,0% исполняемых операторов составили арифметические операторы и операторы присваивания
3,39% исполняемых операторов составили операторы передачи управ» ления
2,3% исполняемых операторов составили операторы CALL
0,4% исполняемых операторов составили вычисляемые GO TO

которую можно было бы записать следующим образом:

```
      A=17
      IF (X .EQ. 0) A = 23
```

Это уплотняет логику программы и к тому же экономит два оператора GO TO. Отметим, что этот пример можно было бы записать на Коболе, ПЛ/1 или Алголе, хотя программист, пользующийся одним из названных языков, скорее всего разрешит эту трудность, написав что-нибудь в роде

```
IF X=0 THEN A=23 ELSE A=17.
```

Помимо этих простых примеров неэффективного программирования, реальная опасность применения оператора GO TO заключена в передачах управления в «разделы программ совместного пользования». Например, наш фрагмент Фортран-программы, вероятно, мог бы быть записан следующим образом:

```
      IF (X .EQ. 0) GO TO 20
      A=17
10    CONTINUE

20    A=23
      GO TO 30

30    CALL GRUMP (I, J, K)
      GO TO 10
```

Как видно из этого, при написании кодов оператора 20 программист вдруг обнаруживает, что ему нужно написать то же самое, что уже было написано по другому поводу в операторе 30. Таким образом, появляется вставка с лишним оператором GO TO, и программист поздравляет себя с экономией двух команд в его программе. К со-

жалению, следить за логикой программы теперь стало намного труднее. К тому же программист может со временем забыть, что в программу оператора 30 можно войти из точки вблизи оператора 20 и незначительными, как кажется, изменениями кодов в операторе 30 вызвать то, что программа распадется на несвязанные части.

Чтобы обезопасить себя от этих осложнений, в некоторых организациях пошли на такую крайность, что запретили использование оператора GO TO в прикладных программах. И если это весьма неудобно в Фортране (из-за слабости логического оператора IF и отсутствия в нем блочной структуры), то вполне сносно в Коболе и представляется естественным подходом в программировании на ПЛ/1 и Алголе.

Отметим здесь же, что если вы, возможно, еще не готовы совсем отказаться от ваших излюбленных операторов GO TO, вы по крайней мере должны отдавать себе отчет в том, что программу, построенную на случайном, хаотичном употреблении операторов GOTO, значительно труднее отладить, чем программу, в основе которой лежат подпрограммы, таблицы решений или какие-нибудь другие модульные структуры. На некоторых из этих идей мы остановимся подробнее еще раз в гл. 4.

Немнемоничность имен переменных. Наконец, мы должны отметить удручающее отсутствие воображения, которое проявляется большинством программистов, когда дело касается выбора имен переменных, меток операторов, имен файлов и записей. Это относится к тем программистам, которые используют имена своих подружек и нецензурные слова (иногда взаимозаменяемо) для различных обозначений. Занятно, когда программист на Коболе умудряется так построить свою программу, что в ней можно прочесть единственное предложение:

```
ADD GIN TO VERMOUTH GIVING MARTINI
```

или когда программист на Фортране записывает

```
00000 = 2 + LOG (3 * 000000)—00000
```

где все переменные 000000 различны по наименованию, но неотличимы по записи, так как быстродействующее печатающее устройство имеет один и тот же символ для *буквы* O и *цифры* 0.

Чаще, однако, встречаются случаи, когда программист использует односимвольные имена переменных, такие, как I, J, K, или такие бессмысленные сокращения, как ORK37. В таких языках, как Кобол, ПЛ/1 и Алгол, программист может употреблять в качестве имен слова, имеющие до 15, 31 или 63 символа. Если программист не страдает неуклюжестью и ленью и готов поработать на телетайпе чуть больше обычного, то эти языки предоставляют достаточные возможности для использования необходимой мнемоники. Вопреки страстным эмоциям некоторых программистов нет ничего

предосудительного, если подпрограмма называется НАЧАЛО-ПРОГРАММЫ или ЧТЕНИЕКАРТНАВВОДЕИПРОВЕРКА-ОШИБОК.

1.2.3. Минимальная стоимость сопровождения

Всякая программа, которая хоть чего-нибудь стоит, будет находиться в обращении в течение долгого времени. Имеется, например, много программ для IBM 1401, первые версии которых были написаны в начале 60-х гг. Впоследствии эти программы имитировались, совершенствовались и основательно перерабатывались в Системах IBM 7040/7044, IBM 7090/7094, IBM/360 и IBM/370, однако первоначальные конструкции и логика остались практически неизменными. Имеется даже несколько программ для IBM 650, которые были написаны в середине 50-х гг. и которые работают по сей день. С другой стороны, встречаются программы, чаще в области научных и технических расчетов, которые используются раз или два и потом выбрасываются. Такие примеры, однако, крайне редки и составляют скорее исключение, чем правило.

Независимо от того, будете ли вы, как первоначальный автор программы, продолжать ее сопровождение или передадите ее кому-то другому, существенно то, что почти всегда последующие дополнения и сопровождение программы необходимы. Лишь совсем недавно в организациях, специализирующихся в обработке данных, начали осознавать значение этих процессов: недавние исследования показали, что в среднем американские организации расходуют на сопровождение50% средств, выделяемых на системы обработкиданных[1]. Другое неофициальное исследование, проведенное в Англии одной крупной организацией, производящей ЭВМ, показало, что программа, написанная отдельным программистом, обычно сопровождается десятью последующими поколениями программистов, прежде чем от нее откажутся полностью и напишут новую.

Существует несколько аспектов проблемы сопровождения; некоторые из них могут быть решены правильным проектированием, другие — нет. Основные проблемы, с которыми сталкиваются в отделе сопровождения, состоят, по-видимому, в следующем:

1. Программы, переданные для сопровождения, все еще содержат значительное число ошибок; таким образом, то, что названо сопровождением, в действительности оказывается продолжением этапа испытаний. Этот факт кажется очевидным, но он часто игнорируется. А в результате программисты, обеспечивающие сопровождение, страдают по вине программистов-разработчиков.

2. Остается проблема модернизации программ при введении новых компиляторов, новых операционных систем и других систем

[1] «That Maintenance Iceberg», EDP Analyzer, October 1972.

математического обеспечения. Маловероятно, чтобы эта проблема была разрешена в ближайшем будущем.

3. Большинство важнейших программ требует постоянного сопровождения в связи с изменениями в составе и потребностях пользователя. Существует очень мало программ, которые были бы так хорошо определены, что не требуют каких-либо изменений.

4. Когда в процессе сопровождения необходимо внести какие-то изменения, оказывается, что программиста-разработчика невозможно найти.

5. Мы сталкиваемся с общей проблемой, состоящей в том, что обычно люди не любят заниматься сопровождением. В этой работе не находят ничего романтического, часто она оплачивается по пониженному тарифу, она сопряжена с повышенным нервным напряжением, так как постоянно приходится сталкиваться с требующими исправления результатами небрежной работы кого-то другого.

6. Серьезная проблема заключается в том, что большинство людей испытывают затруднения в понимании чужих программ. Причина этого, может быть, в том, что большинство программистов характеризуется индивидуальным стилем программирования; главная тому причина, однако, то, что многие программисты пишут свои программы довольно неорганизованно.

7. Описания, сопровождающие большинство программ, ужасны. Некоторые эксперименты показали, что программистам, обеспечивающим сопровождение, лучше с самого начала убрать все комментарии, поясняющие программу, и лишь затем пытаться искать ошибки и вносить улучшения. Ясно, что многие организации сейчас расплачиваются за низкие требования к оформлению программ в прошлом.

Правильное проектирование программ, очевидно, не может разрешить всех проблем сопровождения; однако не менее очевидно, что эти проблемы были бы значительно проще, если бы мы могли проектировать, кодировать и испытывать программы более совершенным путем. Во всяком случае, это предполагает, что при написании программы мы должны организовать ее таким образом, чтобы максимально упростить работу по ее сопровождению. Создавая программу, мы должны постоянно иметь в виду, что ее сопровождение почти наверняка будет выполнено кем-нибудь другим.

Большая часть предложений, содержащихся в настоящей и последующих главах этой книги, должна оказать серьезную помощь программистам, обеспечивающим сопровождение. По всей вероятности, если программа проста в отладке и тестировании, то и ее сопровождение будет относительно легким как для программиста-разработчика, так и для остальных. В этом отношении наиболее важны следующие идеи:

1. Структурное программирование (оно обсуждается в гл. 4) или одна из форм модульного программирования. Чрезвычайно важ-

но избегать хаотичного построения программ; не ясно, сможет ли понять такую программу *программист-разработчик,* но совершенно ясно, что большинство программистов, обеспечивающих сопровождение,— не смогут.

2. Крайне важна простота стиля в программировании. Создавая программу, постоянно спрашивайте себя: «Смог бы это понять рядовой программист, обеспечивающий сопровождение?» Избегайте эгоистической позы — «Всякий приличный программист обязан это понимать...». Не забывайте, что программист, обеспечивающий сопровождение, может не обладать вашими блестящими способностями.

3. «Оформляйте программы для других так, как вы хотели бы, чтобы они оформляли их для вас.» Эта заповедь заимствована из прекрасной книги Kreitzberg, Schneiderman, The Elements of FORTRAN Style, Harcourt, Brace, Jovanovich, 1971.

Разумеется, что, если бы все программисты следовали этим указаниям (так же, как и другим указаниям в этой книге), у нас было бы очень мало трудностей в сопровождении; то, что на самом деле они *не* следуют таким указаниям, является причиной многих неприятностей сопровождения. Это означает, что в действительности мы имеем дело с некоторой задачей организационного управления: отделу сопровожденля не следует принимать программу до тех пор, пока он не сочтет ее отвечающей требованиям. Это, по-видимому, самое слабое место многих организаций в области автоматизированной обработки данных; до тех пор, пока в программе имеются ошибки, с ней должны работать те программисты, которые знают ее свойства, вместо того, чтобы перекладывать эту работу на программистов, которые не понимают программу и не хотят ее знать. Поэтому, прежде чем принять программу к сопровождению, следует провести некоторую проверку того, что она испытана и оформлена в соответствии с некоторыми разумными нормами программирования и *принята* как пользователем,так и отделом сопровождения.

Здесь можно было бы высказать многие другие предложения, касающиеся сопровождения, однако они относятся скорее к области методов организационного управления, которые не являются основным предметом этой книги. С другой стороны, могутбытьнебесполезными некоторые советы тем программистам, которым выпало несчастье сопровождать программу, которая в свое время *не* была спроектирована, испытана и оформлена должным образом. Этот вид работы часто называют сопровождением незнакомой программы. Следующие рекомендации могут оказаться полезными.

1. Прежде чем обратиться к «крайним мерам», изучите предложенную программу. Основательно разберитесь в ней. Если это возможно, найдите автора программы; постарайтесь ознакомиться со всей доступной информацией. Узнайте у автора программы о всех существующих неофициальных описаниях к ней.

2. Попытайтесь разобраться в общей логической схеме управления программой; вначале не обращайте внимания надетали. На этом этапе может оказаться очень полезным построение собственной блок-схемы самого верхнего уровня описания, если таковой нет в наличии.

3. Оцените полезность пояснений, сопровождающих программу. Внесите в листинг свой собственный комментарий там, где вы считаете это уместным.

4. Используйте списки перекрестных ссылок, таблицы символов и другие общие средства, входящие в состав компилятора или ассемблера. Если это необходимо, составьте свои собственные таблицы.

5. Вносите изменения в программу с величайшей осмотрительностью. Насколько это возможно, уважайте стиль и форматы исходной программы. Не превращайте изменение программы в самоцель, например не переписывайте всю подпрограмму ради того, чтобы исключить одну или две бесполезные команды. *На самом листинге всегда отмечайте команды, которые вы изменили.*

6. Исключайте фрагмент из программы лишь в том случае, если вы уверены, что он не используется.

7. Не пытайтесь использовать вспомогательные переменные или рабочую память, уже занятые в программе. Чтобы избежать неприятностей, вводите свои собственные локализованные переменные.

8. Ведите тщательные записи всех изменений, которые вы внесли, всех ошибок, которые вы обнаружили, и всех улучшений, которых вы достигли.

9. Не поддавайтесь неразумному побуждению выбросить исходную программу и написать все заново. Вместо этого ведитетщательные записи, позволяющие оценить вложенные усилия по сопровождению программы; попытайтесь экстраполировать оценки на будущее. Приложите усилия для надежной оценки количества времени, которое вам понадобится для написания новой программы, и помните о том, что ваша программа также потребует сопровождения!

10. Непременно вносите команды контроля ошибок около каждого дописываемого вами фрагмента программы, а также в текст исходной программы всюду, где вы найдете это уместным.

1.2.4. Гибкость программы — в простоте ее изменения, расширения и модификации

Вопреки самым лучшим намерениям, большинство программ в течение времени их жизни изменяется. Требования и технические задания, формулируемые пользователем или заказчиком, редко оказываются «замороженными», так что всегда следует допускать, что вашу программу рано или поздно придется модифицировать,

чтобы сделать ее больше, или быстрее, или более универсальной и т. д.

Этот факт бывает очень трудно внушить начинающим программистам. Очень часто младший программист вступает в спор, утверждая что он пишет «на скорую руку» временную программу, которая не будет использоваться сколько-нибудь продолжительное время. Иногда это так и оказывается, тем не менее он напоминает архитектора, говорящего своему клиенту, что здание, которое он проектирует, мыслится не как долговечное сооружение, а как сколоченная «на скорую руку» постройка. Эта аналогия не такое уж преувеличение, как кажется: некоторые бараки и лачуги, построенные во время второй мировой войны, к большому раздражению современных жителей стоят и по сей день.

Чтобы оценить, легко ли изменить или усовершенствовать программу, вы и ваш руководитель должны постоянно задаваться вопросами: «Что будет, если мы захотим расширить эту таблицу?», «Что произойдет, если однажды мы захотим определить новую программу изменений?», «А что, если нам придется изменить формат таких-то выходных данных?», «Что будет, если кто-то решит вводить данные в программу не с перфокарт, а с телетайпа?».

В большинстве случаев бывает не трудно перечислить соответствующие вопросы; нужно попросту проанализировать программу с той точки зрения, что *всякий* ее элемент может быть изменен, расширен или модифицирован. Таким образом, вам просто надлежит приучить себя к мысли, что *всякий* программный модуль, *всякая* подпрограмма, *всякая* таблица и *всякая* область данных со временем могут быть пересмотрены. Понятно, что, имея в виду эти соображения, довольно трудно решить, как же наилучшим образом писать программу. Здесь, конечно, необходимо установить некоторую систему критериев; может быть, вам следует узнать мнение вашего руководителя или специалиста по системному анализу (если в этих вопросах вы обнаруживаете полную растерянность, то, может быть, вам полезнее посоветоваться со специалистом по психоанализу, чем по системному анализу) прежде, чем решиться на осуществление наиболее общего и гибкого подхода. С другой стороны, очень многие трудности, возникающие в этой области, могли бы быть устранены незначительными усилиями в самом начале проектирования программы. В последующих главах некоторые из этих вопросов, как можно надеяться, станут более ясными.

1.2.5. Минимальные затраты на разработку

Одна из наиболее важных характеристик программы связывается с тем, насколько просто с самого начала спланировать ее разработку. Напомним, что, согласно результатам Сакмана и др., такие **характеристики, как время написания и время испытания программы**

программистом, относятся к числу изменяющихся в наиболее широких пределах. В большинстве случаев фактическое нарушение срока сдачи проекта является наиболее критическим фактором — каждый день отсрочки внедрения программы влечет дополнительные издержки в виде зарплаты программистов, накладных расходов, расходов на содержание аппарата управления; значительные задержки могут привести к выплате неустойки и расторжению контракта стороной, представляющей пользователя программы. Кроме того, нам хотелось бы иметь методы разработки программ для ЭВМ, требующие минимального времени на компиляции, пробные пуски и т. д.

В дополнение к перечисленным обстоятельствам руководитель работ по программированию должен иметь в виду и то, что в течение времени разработки средних и больших программ ведущий программист (по разным причинам) может неожиданно уволиться в связи с женитьбой или уклонением от женитьбы, рождением ребенка, переходом на другую работу или кругосветным путешествием. Таким образом, руководитель должен постоянно задавать себе вопрос, каждый ли программист в группе разрабатывает свои программы так, что его работу мог бы немедленно продолжить кто-нибудь другой; ясно, что хороший программист должен задавать себе этот вопрос *сам*.

Снова напомним.что это лишь принцип; осуществление его на практике обычно не представляет особых трудностей, если его не упускать из виду. Программиста, нанимаемого организациями, разрабатывающими математическое обеспечение и консультирующими в этой области, заставляют привыкнуть к этому правилу довольно быстро: всегда существует опасность (или благоприятная возможность, это зависит от точки зрения), что завтра ему поручат другую работу. Поэтому для успеха проекта, а в некоторых случаях и самой организации, всегда должна быть уверенность, что работу сможет продолжить кто-нибудь другой. Эта предосторожность может показаться излишней кому-нибудь, работающему в более спокойной и устойчивой сфере; однако большинство программистов каждый год или два желают заниматься новой программой, так что самый простой способ — обеспечить возможность продолжения работы над программой силами нового программиста — состоит в учете этого обстоятельства с самого начала разработки программы.

Как и в отношении всех других рекомендаций этой главы, я не был бы удивлен, услышав возражения недовольных, что ваша программа настолько мала и проста, что вам нет необходимости учитывать эти соображения. Не забывайте, что ваша программа, если она хоть чего-нибудь стоит, в течение какого-то времени будет находиться в использовании. Так что, если вам и удастся написать и отладить программу, рано или поздно она будет передана для сопровождения или усовершенствования (и дальнейшей отладки, ко-

торая обычно длится бесконечно) кому-то другому. Один из лучших способов довести это до сознания программиста состоит в том, чтобы поручить ему сопровождение, отладку или модификацию чужой программы; если такой урок преподан программисту в самом начале его карьеры, это оказывает сильное и благотворное действие на его методы программирования.

1.2.6. Простота организации

Во многих отраслях промышленности разработчики установили, что наиболее надежными являются изделия, содержащие минимальное число подвижных элементов. Этот принцип, безусловно, верен и в области производства вычислительной техники: терминалы, лентопротяжные устройства, устройства чтения перфокарт и другие периферийные устройства с минимальным числом движущихся частей представляются более прочными и надежными.

При прочих равных условиях это справедливо и в области математического обеспечения: простой, непосредственный подход, отличающийся минимальным числом специфических особенностей, бывает обычно самым надежным. Например, выбирая между рекурсивной вложенной макрокомандой и простой программой на языке ассемблера, вам было бы выгодно выбрать последнюю; выбирая между компактной Кобол-программой со сложной последовательностью операторов ALTER и GO TO вместо операторов PERFORM и несколько более длинной и медленной, но более простой программой на Коболе, вам выгоднее выбрать последнюю. Выбирая в ПЛ/1 между сложной структурой и более простой, также разумнее предпочесть последнюю.

Можно провести некоторую аналогию с фотоаппаратами. Дорогие «лейка» и «никон» в принципе могут обеспечить очень высокое качество снимков, но какой-нибудь простой аппарат Кодак Инстаматик или Поляроид лучше них во многих отношениях: они намного прочнее и не ломаются, если их уронить на пол или ударить о стену; ваша дочь может залить такой аппарат кока-колой, и он будет работать; вы можете взять его на пляж, не опасаясь засорить песком внутренние механизмы. А самое главное то, что для среднего фотографа в 90% случаев этим аппаратом можно получать снимки не хуже, чем лейкой.

Условие простоты в значительной степени предопределяется другими требованиями предыдущих разделов этой главы. То есть наиболее естественный, а во многих случаях и *единственный,* способ добиться того, чтобы программа была проста в испытаниях, сопровождении или модификации, чтобы работа над ней могла быть легко продолжена кем-нибудь другим, состоит в простом и ясном ее построении.

1.2.7. Эффективность

Большинство программистов обычно считают это свойство наиболее важной характеристикой программы. Многие программисты, включая некоторых из наиболее талантливых и опытных, могут тратить часы и даже дни, пытаясь ускорить подпрограмму на несколько микросекунд, или написать ее на одну команду короче, или уменьшить на одно слово длину массива. Во многих случаях все эта не стоит затрачиваемых усилий.

Если быстродействие действительно важно, вам следует попытаться разумным образом ускорить вашу программу. Вместо того чтобы максимизировать быстродействие каждой подпрограммы, следует сосредоточить усилия на тех подпрограммах, которые применяются наиболее часто. Во многих случаях это может быть выявлено только после того, как программа уже написана, так как характер работы программы может зависеть от вида вводимых в нее исходных данных и способа ее использования заказчиком. Таким образом, лучше всего написать простую программу на Фортране или Коболе, собрать статистические данные по ее эксплуатации и затем переписать на языке ассемблера разделы, определяющие быстродействие программы.

Аналогично, если вы пытаетесь сэкономить память, то это следует делать некоторым разумным образом. Стоит, например, попытаться исключить буфер в несколько тысяч символов или огромный массив размером 5000 X 5000, однако на экономию меньших объемов обычно не стоит тратить много времени. Если память является определяющим фактором, то следует использовать тот же подход — написать программу просто и без ограничений, а затем определить, где можно сэкономить наибольшие объемы памяти. Если эта задача невыполнима, то иногда можно предварительно определить наиболее емкие разделы вашей программы, т. е. разделы, требующие наибольших усилий по оптимизации.

Не следует это воспринимать как совет писать глупые программы. Нетрудно назвать несколько простых методов, позволяющих избегая чрезмерных усилий, писать достаточно эффективные программы. В большинстве случаев этот набор «маленьких хитростей» решает задачу написания эффективной программы; лишь в самых критических случаях, переписывая подпрограмму, следует обращаться к усложненным приемам. Нужно заметить, что, как установлено Кнутом при исследовании Фортран-программ, около 50% времени центрального процессора приходится на 5% команд исходной программы[1)].

1) D. E. Knuth, An Empirical Stady of FORTRAN Programs, Software — Practice and Experience, v. 1, № 2, April-June 1971, p. 105—133.

В отношении эффективности программ может оказаться полезным, если руководитель работ по программированию попытается установить некоторый эквивалент между рабочим временем программиста и машинным временем. Например, некоторые исследования показали, что в среднем программист может написатьот 10 до 20 отлаженных команд в день; другими исследованиями установлено, что при внесении изменений в работающую программу измененная программа работает лишь в 50% случаев. Если программисту платят 12—14 тыс. долл. в год, то с учетом накладных расходов на него тратится около 100 долл. в день. Таким образом, вам и вашему руководителю следует задаться вопросом, сумеете ли вы действительно сэкономить эти 100 долл. на Машинном времени, потратив лишний день на ускорение вашей программы. Такой анализ, однако, может оказаться достаточно опасным, поскольку он показывает вашему руководителю, насколько вдействительности он вам переплачивает. Если ему станет известно, что вы уделяете очень много времени и сил вопросам эффективности, то он может решить понизить вам жалованье!

1.3. Некоторые заключительные замечания относительно „качества" программ

Проведенный в предыдущих разделах анализ носит в основном качественный характер. Подчеркивая, что программа должна быть проста в тестировании и отладке, мы не указали, насколько она должна быть проста, или как измерять это качество. Еще важнее то, что мы ничего не сказали о степени эквивалентности различных желательных свойств программы, которые обсуждались в предыдущем разделе; приведет ли, например, повышение эффективности программы на 10% к увеличению времени испытания на 20%?

В идеальном случае нам хотелось бы иметь возможность приписывать каждой программе некоторое число V, определяющее ее качество. V вычислялось бы по формуле

$$V = a_1(\text{Стоимость испытаний}) + a_2(\text{Стоимость программирования}) + \ldots + a_7(\text{Эффективность}),$$

в которой коэффициенты a_i выбирались бы руководителем (или программистом) для каждой программы. Эти числа должны быть пронормированы, так чтобы V изменялось в диапазоне от 1 до 100; это позволило бы оценивать полезность программы и, конечно, эффективность программиста, который ее написал.

В некоторых приложениях эффективность программы может не иметь существенного значения, и в оценке ее полезности коэффициент a_7 может быть принят значительно большим других коэффициентов. В других случаях мы могли бы придать большее значение затратам на ее испытания, гибкости или легкости сопровождения.

Все это выглядит замечательно, но в настоящее время неприменимо. В некоторыхслучаях у нас нет метода сравнения двух программ в отношении тех характеристик, которые были рассмотрены выше; т. е. мне очень трудно установить, что моя программа в 3,14159 раза лучше вашей в части гибкости. Я могу сказать, что моя программа в 6,7 раза быстрее вашей, или что она в 3,2 раза короче, или что на ее испытание потребовалосьв 2,78128 раза меньше времени.Однако ни одна из этих величин не может быть вычислена прежде, чем программа будет написана. Мы не умеем, взглянув на две блок-схемы, предсказать, которую из них дольше кодировать или испытывать, на которую из них потребуется больше времени центрального процессора (во всяком случае, не умеем с той или иной точностью). У нас нет какого-либо способа сравнивать данную программу с некоторым абсолютом, мы не можем определить, что, скажем, данная программа работает в 3,2 раза медленнее, чем самая быстрая для данного применения программа. Наконец, как ярко показал в своем эксперименте Сакман, мы не знаем, как сравнивать двух программистов, если только мы не предлагаем им разработать одну и ту же программу!

Короче, нам известно очень мало о программах и программистах такого, что может быть измерено или оценено количественно; большая часть того, что мы знаем, состоит из общих принципов и соображений. Определенная работа в этом направлении ведется, но пока нет еще результатов, о которых можно было бы рассказать. А пока лучшее, что мы можем сделать,— передать свой опыт другим программистам и надеяться, что когда-нибудь явится какой нибудь Галилей или Ньютон и превратит в науку лучшие приемы того колдовства, которое мы называем программированием для ЭВМ.

ВОПРОСЫ

1. Дайте ваше собственное определение хорошего программиста. Следует лн исходить из одних и тех же требований к программисту при найме его на работу и при рассмотрении вопроса о значительном повышении жалованья?

2. Как, на ваш взгляд, определяет хорошего программиста ваш руководитель? Сильно ли отличается его определение от вашего? Имеет ли он представление о вашем определении? Обсуждали ли вы с ним эту тему?

3. В чем состоят выдающиеся способности самого лучшего из известных вам программистов? Можно ли, на ваш взгляд, эти способности развить обучением или научиться передавать их другим? Считаете ли вы, что некоторые из этих способностей основаны на опыте? Или они являются врожденными?

4. Как может быть измерена квалификация хорошего программиста? В чем недостаток и ограниченность использования «числа отлаженных команд исходной программы в день» в качестве такой меры?

5. Дайте ваше собственное определение хорошей программыдля **ЭВМ.** Можно ли создать такую программу в обычных рабочих условиях?

6. Насколько важной вы считаете эффективность программы в сравнении **с** такими характеристиками, как простота сопровождения, . . . и т. д.?

7. Возьмите какой-нибудь счет, составленный с помощью ЭВМ (например, **местной телефонной компании, налогового управления и т. д.). Очевидно ли, что**

указанная сумма (в частности, величина налога, стоимость кредита, стоимость услуг и т. д.) правильная? Укажите, что бы вы сделали для того, чтобы правильность выходных данных была очевидной любому человеку.

8. Помимо соображений, перечисленных в настоящей главе, можете ли вы назвать какие-нибудь доводы в пользу программ без комментариев в листинге?

9. (Эксперимент.) Возьмите достаточно хорошо прокомментированную программу, незнакомую вам и нескольким вашим коллегам приблизительно равной квалификации в программировании. Хорошим примером такой программы может служить программа, пакет программ, операционная система или компилятор, поставляемые вместе с ЭВМ. Задайтесь целью внести некоторое изменение в программу, например некоторое ее улучшение или расширение, или выявить некоторую ошибку. Одной группе программистов дайте для работы вариант программы без комментариев, другой — исходную прокомментированную программу. Ведите дневник и статистику, по которым могут быть сделаны выводы о действительной пользе комментариев в листинге программы,

10. Напишите программу, которая может анализировать исходные программы и определять плотность комментариев в них (например, число комментариев на одну команду, или на одну подпрограмму, или один модуль исходной программы). Проанализируйте достаточно большой набор программ, например все библиотечные программы вашей организации или все библиотечные программы поставляемого математического обеспечения. Какова средняя плотность комментариев? Каковы вариации в плотности от программы к программе? (Автору было бы интересно познакомиться с любой статистикой, собранной в этой области.)

11. Можете ли вы привести какие-нибудь доводы, кроме более высокой эффективности, которыми оправдывается использование языка ассемблера вместо языков программирования высоких уровней? Если да, то попытайтесь оценить степень важности этих доводов.

12. Считаются ли в вашей организации опытные программисты, работающие на языке ассемблера, более квалифицированными, чем те, которые знают только один язык программирования высокого уровня? Имеют ли они более высокое жалованье? Считаете ли вы это справедливым? Насколько высоко ценятся программисты, которые знают два и более языков программирования высокого уровня?

13. Обладает ли ваша вычислительная система какими-нибудь чертами многоцелевого использования (например, через макрокоманду ATTACH в Системе IBM/370)? Установите, сколько на это расходуется дополнительного времени центрального процессора. Какие случаи программирования оправдывают эти потери? Приведите пример, когда многоцелевое использование применимо, но экономически не оправдано.

14. Назовите три причины, по которым многоцелевое использование может легко привести к затруднениям в испытаниях. Если вы находите это сложным, обратитесь к книгам E. Yourdon, Design of On-Line Computer Systems, Prentice-Hall, 1972, или J. Martin, Design of Real-Time Computer Systems, Prentice-Hall, 1967.

15. Приведите другой пример сложной ситуации, вызванной одновременным обращением двух программ к одному и тому же файлу. Если вы находите это трудным, обратитесь к гл. L книги E. Yourdon, Design of On-Line Computer Systems.

16. Дайте описание пяти примеров хитроумного, извращенного, непредусмотренного и нечестного программирования на языке Фортран. Можете ли вы назвать ситуации, в которых такая практика оправдана? В тех случаях, когда вам доводилось видеть (или использовать) подобные приемы, сопровождались ли они соответствующим комментарием, предупреждающим неосмотрительного программиста, обеспечивающего сопровождение, о точном смысле использованных кодов?

17. Дайте описание пяти примеров хитроумного, извращенного, непредусмотренного и нечестного программирования на языке Кобол. Можете ли вы назвать ситуации, в которых такая практика оправдана? В тех случаях, когда вам

доводилось видеть (или использовать) подобные приемы, сопровождались ли они соответствующим комментарием, предупреждающим неосмотрительного программиста, обеспечивающего сопровождение, о точном смысле использованных кодов?

18. Дайте описание пяти примеров хитроумного, извращенного, непредусмотренного и нечестного программирования на языке ПЛ/1. Можете ли вы назвать ситуации.в которых такая практика оправдана?В тех случаях, когда вам доводилось видеть (или использовать) подобные приемы, сопровождались ли они соответствующим комментарием, предупреждающим неосмотрительного программиста, обеспечивающего сопровождение, о точном смысле использованных кодов?

19. Дайте описание пяти примеров хитроумного, извращенного, непредусмотренного и нечестного программирования на языке Алгол. Можете ли вы назвать ситуации, в которых такая практика оправдана? В тех случаях, когда вам доводилось видеть (или использовать) подобные приемы, сопровождались ли они соответствующим комментарием, предупреждающим неосмотрительного программиста, обеспечивающего сопровождение, о точном смысле использованных кодов?

20. Дайте описание пяти примеров хитроумного, извращенного, непредусмотренного и нечестного программирования на языке ассемблера. Можете ли вы назвать ситуации, в которых такая практика оправдана? В тех случаях, когда вам доводилось видеть (или использовать) подобные приемы, сопровождались ли они соответствующим комментарием, предупреждающим неосмотрительного программиста, обеспечивающего сопровождение, о точном смысле использованных кодов?

21. Дополните анализатор исходных программ, описанный выше в упражнении 10, средствами контроля применения в программе порочных методов программирования, перечисленных в п. 16—20. Проанализируйте несколько библиотечных программ вашей организации и установите, насколько широко используются такие методы; сравните это с анализом программ ассоциации пользователей ЭВМ типа вашей (например, SHARE, DECUS, CUBE и т. д.). Не находите ли вы, что проведение подобного анализа следует считать необходимым условием того, чтобы позволить программисту внести его программу в библиотеку?

22. Какие ограничения следовало бы определить в отношении программ, которые сами себя модифицируют, например, с помощью оператора ALTER в Коболе? Было бы разумным вовсе запретить использование таких операторов?

23. В программах, написанных вами (или, если это затруднительно, написанных кем-нибудь другим), найдите примеры использования переменных в качестве общей рабочей памяти двух или более разделов программы. Привело ли это к каким-нибудь сложностям? Почему вы написали программу таким образом — чтобы сэкономить память или просто из-за лени?

24. Обсудите те особенности языка Фортран, которые представляются естественными или неудобными при использовании программистом общих массивов памяти для хранения промежуточных данных.

25. Обсудите те особенности языка Кобол, которые представляются естественными или неудобными при использовании программистом общих массивов памяти для хранения промежуточных данных.

26. Обсудите те особенности языка ПЛ/1, которые представляются естественными или неудобными при использовании программистом общих массивов памяти для хранения промежуточных данных.

27. Обсудите те особенности языка Алгол, которые представляются естественными или неудобными при использовании программистом общих массивов, памяти для хранения промежуточных данных.

28. Обсудите те особенности языка ассемблера, которые представляются естественными или неудобными при использовании программистом общих массивов памяти для хранения промежуточных данных.

29. Расположите упомянутые пять языков по степени убывания надежности контроля над неосторожным использованием программистом одних и тех же переменных или рабочей памяти в двух или более модулях^программы.

30. Можно ли оведением строгих норм программирования снять эту проблему общих массивов рабочей памяти?

31. Можно ли создать препроцессор (или анализатор), который мог бы контролировать программы на предмет использования в них общих массивов памяти временного хранения? Если да, то насколько это было бы трудно? Стоит ли прилагать такие усилия?

32. (Эксперимент.) Попытайтесь составить макрокоманду, которая корректна (в соответствии с руководством по языку ассемблера, составленным поставщиком ЭВМ), но настолько сложна, что

а) ассемблер зацикливается или останавливается;

б) ассемблер формирует ошибочную программу. Если это вам удалось, то

а) направьте сообщение об ошибке поставщику ЭВМ;

б) принесите извинения группе операторов ЭВМ;

в) исследуйте возможные практические применения вашей сложной макрокоманды.

Во всяком случае, постарайтесь определить, сколько времени центрального процессора потребовалось ассемблеру на операции по разбору вашей макрокоманды и формированию объектной программы.

33. Напишите программу, способную определять типы команд, используемых в программах (см. обсуждение программы Кнута в разд. 1.2.2). Используйте ее для анализа достаточно большого набора программ в вашей организации. Вот некоторые аспекты, которые можно исследовать:

а) Сколько в среднем операторов GO TO используется в программах?

б) Наблюдаются ли значительные вариации от программиста к программисту в использовании различных типов операторов?

в) Наблюдаются ли значительные вариации между различными языками программирования в использовании основных типов операторов (например, операторов GO TO и операторов вызова подпрограмм)? Тоесть,больше ли таких операторов в средней Фортран-программе, чем в средней Кобол-программе?

г) Наблюдается ли устойчивая связь между числом операторов GO TO и

1) эффективностью программы (частое оправдание применения оператора GO TO)?

2) модульностью программы?

3) количеством тестовых пусков, которое потребовалось при доведении программы? (Если вы не знаете, как определить это количество, см. гл. 7.)

4) адекватностью комментариев?

34. В разд. 1.2.2 рассматривается простой пример программы, составленной на Фортране, и делается вывод, что некоторые операторы GO TO могут быть исключены, если употребить следующую запись:

```
A=17
IF (X .EQ. 0) A=23
•
```

В чем состоят возможные неудобства такого подхода? Считаете ли вы, что такой стиль следует закрепить как норму программирования на Фортране? Почему да или почему нет?

35. Дополните анализатор исходных программ, описанный в п. 10 и 21, блоками определения длины имен переменных и других меток в программах, написанных в вашей организации. Вы могли бы изучить следущие два взаимосвязанных вопроса:

а) Какова средняя длина меток? Сильно ли изменяется длина метки от программиста к программисту?

б) Некоторые языки программирования, особенно Фортран и многие языки ассемблера, допускают длину переменной, не превышающую шести или восьми

символов. В какой мере используется эта возможность программистами? Например, если средняя длина употребляемых меток равна четырем символам, а максимально допустимая длина — восьми, то используется 50% имеющихся возможностей. Имеет ли это большое значение?

36. Считаете ли вы разумным требование, чтобы все метки имели длину не менее чем *N* символов? Каким должно быть соответствующее значение *N?* Можно ли расширить анализатор исходных программ, описанный в п. 10, 21 и 35, таким образом, чтобы он смог фиксировать нарушение этого требования к меткам? Желателен ли такой контроль?

37. В 1971 году мне представился случай обсудить вопросы мнемоники и содержательности меток с программистом, разрабатывающим важные имитационные программы для Центра пилотируемых космических кораблей НАСА. По его мнению, переменным, подпрограммам и другим меткам не следует давать «значащие» имена, поскольку впоследствии они могут быть ошибочно поняты или истолкованы программистами, обеспечивающими сопровождение. По этой причине он намеренно выбирал такие имена, как QRK17, GLOP42 и ZYX123, ни одно из которых не имело никакой связи с именуемыми переменными и подпрограммами. Он считал, что в этом случае программист будет вынужден тщательно изучить программу, чтобы узнать истинный смысл этих меток. Согласны ли вы с таким принципом? Считаете ли вы, что он имеет определенное преимущество?

38. Какая доля средств, выделяемых на обработку данных, расходуется в вашей организации на сопровождение? Сколько программ в настоящее время ведется отделом сопровождения? Сколько лет самой первой программев вашей библиотеке? (Не удивляйтесь, если, обратившись к соответствующим компетентным лицам, вы получите эту информацию не сразу. Многие организации просто не знают, сколько они расходуют средств в этой области.) Говорит ли это вам что-нибудь о важности сопровождения и о важности написания программ таким образом, чтобы их сопровождение было более простым?

39. (Эксперимент.) Найдите среди ваших коллег энтузиаста и выберите из любой главы этой книги две задачи, в которых требуется написать программу. Одну из задач возьмите сами, другую дайте вашему коллеге. Продолжайте работу до наступления одного из следующих ключевых событий:

а) Закончено проектирование высшего уровня описания (более детальное обсуждение нисходящего проектирования найдете в гл. 2).

б) Закончено детальное проектирование.

в) Закончено написание программы.

г) Программа транслируется (или компилируется) и не содержит серьезных ошибок.

д) Правильно решается один контрольный вариант.

В этот момент вам следует обменяться программами, т. е. вы должны будете закончить программу, начатую вашим коллегой, а он — начатую вами. Тщательно фиксируйте все трудности, с которыми вы сталкиваетесь, заканчивая программу партнера. Что мог бы сделать партнер, чтобы облегчить вашу работу? Что могли бы сделать вы, чтобы облегчить его работу?

Вариант. Начните работу над задачами так, как это описано выше, но не решайте заранее, в какой момент следует обменяться программами. Третий участник (арбитр) следит за продвижением обеих программ и без предварительного предупреждения принимает решение об обмене программами в тот самый момент, когда это может вызвать наибольшее замешательство. (Таким образом имитируется обычная ситуация в области разработки программ!)

40. Напишите короткое эссе, опровергающее принцип простоты в проектировании и написании программ. Вы действительно верите в то, что вы пишите?

4t. Выберите небольшой фрагмент написанной вами очень «умной» или изощренной программы; он должен быть достаточно коротким — не более страницы. Покажите программу пяти вашим способным коллегам'и запишите, сколько времени потребуется на то, чтобы каждый из них:

а) понял вашу программу;

б) успешно внес нетривиальное изменение или улучшение в вашу программу (изменение, которое вы должны точно определить).

Велико ли различие между наибольшей и наименьшей работой, проделанной вашими коллегами? Говорит ли это вам что-нибудь о достоинствах простоты?

42. Напишите короткое эссе, излагающее вашу собственную точку зрения по следующим вопросам:

а) Как следовало бы измерять качество программы?

б) Почему сегодня мы не можем этого делать?

в) Какую работу нужно проделать, чтобы определить разумный набор количественных критериев для оценки программ?

ГЛАВА 2

НИСХОДЯЩЕЕ ПРОЕКТИРОВАНИЕ ПРОГРАММ

2.0. Введение

Теперь, когда мы закончили обсуждение некоторых отличительных особенностей хороших программ для ЭВМ, представляется уместным задать вопрос. *Каким, образом нам следует проектировать программы?* Существует ли такая регулярная процедура, используя которую мы могли бы организовать решение задачи программирования, представив его в форме, понятной как вычислительной машине, так и программисту сопровождения?

Суть этой главы, как и большей части всей книги, состоит в доказательстве утверждения, что соответствующий рациональный подход к проблеме существует; в области автоматизированной обработки данных его называют нисходящим проектированием (или проектированием *сверху вниз).* Этот подход интуитивно привлекателен, за последние несколько лет он подвергался неоднократному обсуждению в литературе. Нисходящее проектирование известно и под другими названиями, например «конструктивное программирование» [1], «программирование пошаговым совершенствованием» [2], «систематическое программирование» [3] и «иерархическое проектирование» [4].

Интересно отметить, что многие организации, специализирующиеся в области автоматизированной обработки данных, предприняли попытки выработать свое собственное представление о том, что можно было бы назвать «канонической формой» *проекта* программы, (каноническая форма организации внутренней логики программы, известная под названием структурного программирования, обсуждается в гл. 4). В этих организациях заметили, что многие прикладные программы выполняют однотипные операции и, следовательно, они должны быть подобными по форме и структуре. Так, в некоторых организациях решили: «Все наши программы имеют некоторый инициирующий модуль, основной набор модулейчтения/записи, некоторый модуль основной обработки и набор терминальных подпрограмм и операций, следовательно, все наши программы должны быть организованы таким образом, чтобы все эти модули были формально различимы».

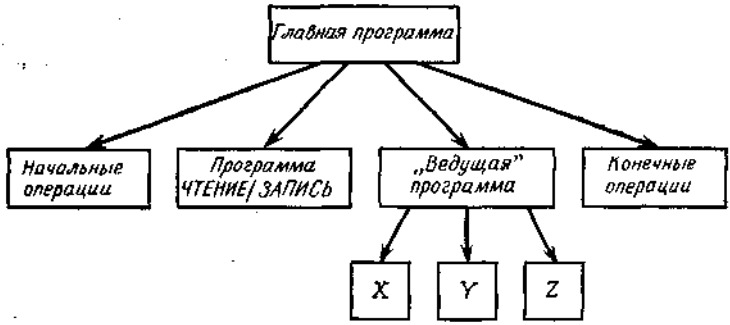


Рис. 2.1. Стандартная форма организации программы.

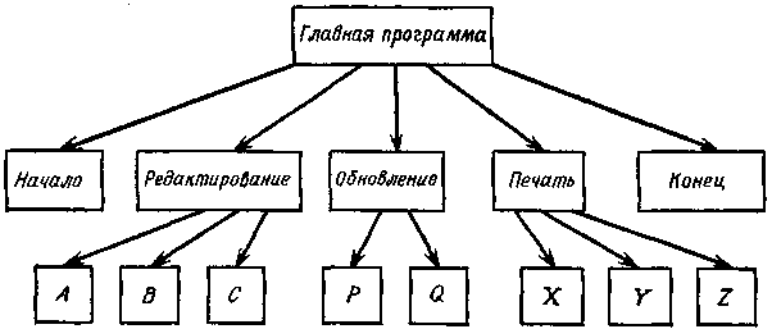


Рис. 2.2. Стандартная форма организации экономических программ.

В результате программисты часто получали директивное предписание представлять их программы в форме, показанной на рис. 2.1; при этом одна прикладная программа отличается от другой лишь модулями нижнего уровня (обозначенными на рис. 2.1 через *X, Y, Z)*. В других организациях пошли дальше и решили: «Все наши экономические прикладные программы включают редактирование (и сортировку) сообщений, или файлов изменений, за которым следует обновление главного файла и печать одного или нескольких отчетов; следовательно, наши программы должны оформляться так, чтобы все эти модули были различимы по формату». Программистам в этих организациях предписывается оформлять всякую программу в соответствии с общей структурой, представленной на рис. 2.2. Как и прежде, прикладные программы отличаются одна от другой только модулями более низкого уровня (X, *Y,* Z и т. д.).

Другим организациям, работающим в области автоматизированной обработки данных, повезло в меньшей степени; не все их прикладные программы попали в столь узкую категорию. Тем не менее и здесь были предприняты попытки формализовать проектное описание программ. Это основывалось на уверенности в том, что большая

часть этих программ характеризуется *определенным* сходством, хотя и в меньшей степени, чем сходство между почти идентичными программами типа редактирование — обновление — сортировка — печать, которые представлены схемой на рис. 2.2. Если смотретьшире, *то* можно сказать, что в этих организациях пытались поощрять у программистов стремление следовать общему принципу, который называли такими словами, как «магистральное проектирование», «взрывное проектирование» и т. д.

Методология проектирования, которую мы предлагаем в этой главе, во многом сходна с этими специальными методологиями, разработанными в отдельных организациях в связи с их конкретными прикладными задачами. Основные принципы, на которые опирается нисходящее проектирование, примениимы к *любой* прикладной задаче; в специальных случаях они могут гриниматьформу, отвечающую схемам рис. 2.1 или 2.2. Не так важно издать приказ типа «Все программы, создаваемые в данной организации, *должны* отвечать формату редактирование — обновление — сортировка — печать», как понять те принципы, которые лежат в основе такой канонической формы. Без этих руководящих принципов программист может проектировать модули нижнего уровня хаотично, в результате чего вся программа, подобно творению Франкенштейна, являет собой произвольное объединение разумной структуры верхнего уровня описания и неорганизованного, плохо спроектированного тела программы. Поскольку большая часть кодирования приходится на тело программы, важно, чтобы принципы, положенные в основу схем рис. 2.1 и 2.2, могли быть перенесены на более низкие структурные уровни программ.

Идеи нисходящего проектирования более детально рассматриваются в следующем разделе. Далее обсуждаются близкие к этим идеям принципы *нисходящего кодирования* и *нисходящего тестирования*. Следует иметь в виду, что все эти идея излагаются как *принципы* или *стратегии,* однако такие, которые, как мы надеемся, способны служить инструментами проектирования программы. Важно осознать, что эти принципы не являются универсальным законом природы; не следует их считать и десятью заповедями некой новой религии. В действительности в конкретных ситуациях здравый смысл всегда подскажет те или иные варианты основных принципов нисходящего проектирования. Некоторые из таких вариаций обсуждаются в разд. 2.4. Наконец, с целью иллюстрации принципов в разд. 2.5 рассматривается учебный пример.

2.1. Нисходящее проектирование

Одним из естественных подходов к проектированию программ является подход *сверху вниз.* Существуя под разными названиями, такими, как «систематическое программирование», «иерархическое

проектирование программ» и «взрывное проектирование», большинство различных вариаций нисходящего проектирования преследует одну и ту же цель: определить основные функции, которые должны быть обеспечены, а затем перейти к определению дополнительных функций, которые вытекают из этих основных. Хотя основной принцип весьма прост, необходимо, как это будет показано ниже, иметь в виду многие особенности его приложения.

2.1.1. Основная идея нисходящего проектирования

Основная идея нисходящего проектирования *действительно* проста, по крайней мере в теории. Рассмотрим какую-нибудь (произвольную) программу, назовем ее GLOP. Начиная проектировать нашу программу, представим себе, что у нас имеется вычислительная машина, в которой осуществима аппаратная реализация операции GLOP (не столь уж нереальная идея при современном уровне микропрограммирования ЭВМ!). Таким образом, чтобы написать программу GLOP, мы просто пишем команду

```
GLOP
```

и все готово! Программа не только написана, но мы почти уверены, что она составлена правильно, если, конечно, примитив с именем GLOP выполняется правильно (если это не так, то мы всегда можем упрекнуть в этом поставщика ЭВМ).

К сожалению, не многие современные ЭВМ имеют в составе системы команд операцию GLOP; поэтому нам придется расчленить задачу, а следовательно, и программу, на более мелкие элементы. Нам представляется, что исполнение GLOP включает некоторые инициирующие процедуры, какие-то вычисления и некоторые заключительные действия (например, закрытие файлов и печатание выходных данных). Таким образом, мы переписываем нашу программу GLOP в следующем виде:

```
НАЧАТЬ GLOP
ВЫПОЛНИТЬ ВЫЧИСЛЕНИЯ GLOP
ЗАКОНЧИТЬ GLOP
```

при этом НАЧАТЬ GLOP, ВЫПОЛНИТЬ ВЫЧИСЛЕНИЯ GLOP и ЗАКОНЧИТЬ GLOP снова могут рассматриваться как простейшие команды некоторой гипотетической машины, для которой мы программируем. Затем этот процесс продолжается: каждая из определенных выше операций расчленяется на более простые до тех пор, пока, наконец, мы не придем к таким примитивам, или элементарным операциям, которые настолько малы и просты, что могут быть аппаратно реализованы на одной из *существующих* вычислительных машин.

2.1.2. Пример нисходящего проектирования

Чтобы продолжить иллюстрацию идеи нисходящего проектирования, мы рассмотрим несколько шагов в проектировании информационной системы управления, которая часто упоминается под названием *финансовой модели*. Представим себе дилемму, стоящую перед менеджером, пытающимся решить, стоит ли ему вкладывать средства в новое дело. Ему необходимо знать, как велика будет ежемесячная прибыль, какую сумму наличных денег он должен вложить в это дело прежде, чем он начнет получать прибыль, и т. п. Все эти данные должны быть получены на основе оценок числа продавцов, нанимаемых менеджером для нового торгового предприятия, числа клиентов, которых удастся привлечь этим торговым агентам, суммы выплат, которые могут быть получены от этих клиентов, расходов, связанных с производством, распределением товара и сопутствующими услугами, и т. д. Все это может быть реализовано в виде программы, составляющей то, что называют *имитационной* моделью нового предприятия.

Краткое описание одной из таких имитационных моделей системы с разделением времени, предназначенной для информационного обслуживания торгового предприятия (где автор приходит к выводу, что сама по себе система с разделением времени убыточна независимо от числа клиентов!), можно найти в работе [5]. Более подробное описание этой задачи дается в приложении А настоящей книги. Эта задача использовалась в качестве примера в нескольких курсах программирования по всем Соединенным Штатам, в Европе и Австралии; более 35 различных групп проектировщиков пытались разрешить ее множеством различных способов.

Допустим, что перед нами стоит задача спроектировать и написать такую программу в соответствии со спецификациями, перечисленными в приложении А; за отсутствием лучшего имени мы решили назвать эту программу ДЕНЬГИ. Как мы уже видели, на первом шаге нам следует вообразить, что существует машина, в системе команд которой имеется команда ДЕНЬГИ. В таком случае, чтобы написать программу, мы просто пишем

```
ДЕНЬГИ
```

Поскольку, однако, не похоже, чтобы в реальном плане это продвинуло нас слишком далеко, мы должны предпринять в нашем проектировании следующий шаг. Нам представляется, что для моделирования финансовых отношений, интересующих менеджера, мы должны определить некоторые *входные параметры,* значениями которых являются.такие величины, как задаваемые менеджером оценки числа нанимаемых торговых агентов, их производительности, ожидаемый доход от каждого покупателя и т. д.; конкретно в приложении А перечисляется до 48 различных входных параметров. Исходя

```
COMMENT ГЛАВНАЯ ПРОГРАММА)
BEGIN
INITIALIZETHEWORLD;
READPARAMETERCARDS;
EDITPARAMETERCARDS;
IF CARDSAREOKFLAG THEN
     WHILE MOREPARAMETERSTOWORKONFLAG
          BEGIN
             PRINTREPORTS;
             EVALUATEPARAMETERS;
          END;
ELSE PRINTERRORMESSAGE;
FINISHOFFEVERYTHING;
END;
```

Рис. 2.3а. Структура логики верхнего уровня программы ДЕНЬГИ.

из техзданий можно заметить, что у менеджера может возникнуть желание изменить те или иные параметры некоторым образом, фактически весьма сходным с тем, как это делается в Фортране оператором цикла DO: он может исследовать поведение прибыли и поступление денежной наличности при пяти торговых агентах; затем он может пожелать исследовать ту же ситуацию с неизменными значениями всех параметров, но с увеличением до 10 числа торговых агентов; он может пожелать повторить те же вычисления для случая с 15 торговыми агентами и т.д., скажем вплоть до варианта с 50 торговыми агентами.

Таким образом, мы можем представить, что наша программа начинается с выполнения некоторых инициирующих действий (точная природа которых к данному моменту еще не ясна). Затем программа прочтет карты со значениями параметров, которые должны быть отредактированы, чтобы убедиться в том, что в них нет неисправимых ошибок. Если это так, то далее нам следует выполнить наши основные вычисления (например, значений числа покупателей, доходов, затрат и прибылей); эти вычисления должны повторяться до тех пор, пока мы не исчерпаем всех возможностей в вариациях входных параметров. Таким образом, мы приходим к программе, приведенной нарис.2.3а. К сожалению, многие слушатели курсов программирования, руководимых автором, высказались против иллюстрации программ в терминах Алгола, и мы часто вынуждены записывать ее на Коболе, как показано на рис. 2.3б. И, конечно же, можно заметить, насколько велико различие на этом этапе!

В каком направлении нам следует продвигаться теперь? Идеально, нам хотелось бы иметь такую машину, в которой все операторы, приведенные на рис. 2.3а или 2.3б, могли быть реализованы

```
MAIN-PROGRAM SECTION.
PERFORM INITIALIZETHEWORLD.
PERFORM READPARAMETERCARDS.
PERFORM EDITPARAMETERCARDS.
IF CARDSOKFLAG=1 THEN
   PERFORM PROCESSCARDS
         UNTIL MORECARDSFLAG = 0
ELSE
     PERFORM PRINTERRORMESSAGE.
PERFORM FINISHOFFEVERYTHING.
STOP RUN.
```

Рис. 2.3б. Описание на Коболе структуры логики верхнего уровня программы ДЕНЬГИ.

в виде машинных команд. Очевидно, однако, что предстоит выполнить еще несколько этапов проектирования до того, как мы найдем такую машину. Итак, нам следует продолжать в том же духе. Каждый оператор, приведенный на рис. 2.3а или 2.3б, следует представлять как некую абстракцию, которую мы должны попытаться реализовать с помощью простейших команд более низких уровней. Не всегда, однако, следует это делать в строго последовательном порядке, как это видно из следующего примера: пока мы не знаем, что именно следует инициировать, нет смысла уточнять оператор INITIALIZE (НАЧАТЬ). Мы можем в то же время попытаться расписать оператор READPARAMETERCARDS (ЧТЕНИЕКАРТПАРАМЕТРОВ), так как, вероятно, мы должны будем считать все 48 карт и организовать их в виде некоторой таблицы для дальнейшего использования.

Точно так же оператор PRINTREPORTS (ПЕЧАТЬОТЧЕТОВ), приведенный на рис. 2.3а и 2.3б, может быть расписан в форме, показанной на рис.2.4а, или 2.4б. Заметим, что основные вычисления мы выполняем с месячным интервалом, начиная с месяца номер 1 и продолжая их до тех пор, пока не настанет месяц, на котором в соответствии со спецификациями следует остановиться. В этой точке мы вызываем другой оператор, EVALUATEPARAMETERS (ОЦЕНКАПАРАМЕТРОВ), который выясняет, имеются ли еще неопробованные сочетания параметров.

Представляет определенный интерес рассмотреть, как действовали в этой ситуации слушатели моих курсов программирования; стоит отметить, что большая их часть имела опыт работы в области автоматизированной обработки данных сроком 2—15 лет. В большинстве случаев структура программы верхнего уровня описания, представленная рис. 2.3а (или 2.3б) и рис. 2.4а (или 2.4б) принималась в качестве разумной отправной точки. С этим начальным состоянием слушатели отправлялись в учебные аудитории, имея за-

```
PROCEDURE PRINTREPORTS;
BEGIN
INTEGER  l;
FOR I: 1 STEP 1 UNTIL N47 DO
     BEGIN
       CALCULATESALESMEN  (I);
       CALCULATECUSTOMERS  (I);
       CALCULATEREVENUE  (I);
       CALCULATECOSTS  (I);
       CALCULATEPROFITLOSS  (I);
     END:
END PRINTREPORTS;
```

Рис. 2.4а. Расширение программного кода верхнего уровня, представленного на рис. 2.3а.

```
PROCESSCARDS SECTION.
   PERFORM  BASIC-CALCULATIONS  VARYING
      T FROM  1  BY  1  UNTIL  T GREATER
      THAN  N47
   PERFORM  EVALUATEPARAMETERS.
   EXIT.
BASIC-CALCULATIONS.
   PERFORM  CALCULATESALESMEN.
   PERFORM  CALCULATECUSTOMERS.
   PERFORM  CALCULATEREVENUE.
   PERFORM  CALCUCATECOSTS.
   PERFORM  CALCULATEPROFITLOSS.
   PERFORM PRINTREPORTS.
   EXIT.
```

Рис. 2.4б. Расширение программного кода верхнего уровня, представленного на рис. 2.3б.

дание завершить проектирование и получить программную реализацию решения этой задачи.

Естественно, что их внимание сосредоточилось на вычислениях, выполняемых модулем BASICCALCULATIONS (ОСНОВНЫЕВЫЧИСЛЕНИЯ), так как именно здесь, как было отмечено всеми, таиться суть проблемы. Поскольку первым подмодулем в модуле ОСНОВНЫЕВЫЧИСЛЕНИЯ является CALCULATESALESMEN (ОЦЕНКАЧИСЛАТОРГОВЫХАГЕНТОВ), им, естественно, потребовалось установить, каким образом вычислить количество торговых агентов предприятия. Беглый анализтехзаданияпоказывает, что определенное число торговых агентов нанимается в гипотетическое торгово.е предприятие в момент его возникновения; до-

полнительное число торговых агентов будет принято в последующие месяцы, пока их общее количество не достигнет некоторого максимального числа; и наконец, спецификациями предусматривается, что определенный процент торговых агентов будет увольняться с предприятия в любой рассматриваемый месяц.

По какой-то причине внимание обучающихся привлекала именно та часть спецификаций, которая касается числа торговых агентов, которые могут уволиться. То, что число увольняющихся составляет некоторый *процент,* оказывает на многих сильное впечатление; они отмечают, что, если не ввести особые меры предосторожности, это может означать возможность увольнения *части* одного торгового агента. Поскольку агенты — это люди и поскольку число людей должно быть целым, учащимся представляется ужасным, если, например, согласно их финансовой модели к концу первого месяца существования предприятия уволится 3,14159 торгового агента. Многие в этом случае решаются на предположение, что часть торгового агента, который желает уволиться в месяц номер N, должна переноситься на месяц номер $N+1$, где она может быть скомбинирована с другими такими же частями, составляя таким образом целого агента, который и может уволиться с предприятия. Естественно, что это приводит к дискуссии о том, какими средствами следует описывать число агентов в программе. Следует ли использовать числа с плавающей точкой и обычной точностью, или с плавающей точкой и двойной точностью, или целые? А как быть с младшими разрядами? Следует ли округлять результаты всех вычислений, включающих число торговых агентов?

Хотя вся эта ситуация может показаться довольно нелепой, часто случалось, что группа программистов спорила *часами* именно по этой части предложенного упражнения. На самом деле предмет спора часто переносится на программистов и других сотрудников, которые при некоторых обстоятельствах могут уволиться из фирмы. Снова возникают вопросы: «Как оперировать сдолями программиста?» и «Каким образом могут уволиться 2,78128 программиста?». Вполне реально, что кто-то, имея соответствующие политические убеждения, займет крайнюю позицию и будет настаивать на том, что программистов вообще не следует увольнять! После двух часов горячих споров на эту тему в одной из моих групп предметом дебатов оказались вопросы восточной религии и альтернативных форм социальной организации; до следующего дня учащихся уже невозможно было вернуть к обсуждению столь приземленных предметов, как программирование!

Этим программистам, кажется, не пришло в голову, что в рамках финансовой модели пользователю, скорее всего, нет дела до дробных частей программистов; им, кажется, не пришло в голову, что в числе программистов (или торговых агентов) можно отбрасывать младшие разряды, что эти числа могут округляться по из-

бытку или недостатку. Им, очевидно, не пришло в голову, что, каков бы ни был их выбор, пользователь может оказаться неудовлетворенным и попросить внести изменения (например, округление по избытку вместо округления по недостатку). Главная идея в данном случае состоит в следующем. *Представляется ли число агентов в виде переменной с плавающей точкой и обычной точностью или нет, совершенно не существенно на рассматриваемой стадии проектирования!* На каком-то из последующих этапов ее, очевидно, придется разрешить, но пока это самая незначительная из задач проектирования.

Если программистам группы становится это ясным (или, что более вероятно, если они наконец приходят к какому-нибудь предварительному согласию относительно характера этой переменной), то они продолжают проектирование алгоритма модуля ОСНОВНЫЕВЫЧИСЛЕНИЯ. Им приходит на ум попытаться выразить число торговых агентов с помощью какой-нибудь математической формулы, записав, например, соотношение, подобное оператору Фортрана,

ЧИСЛОТОРГОВЫХАГЕНТОВ = A+ B + C + D

Очень скоро, конечно, они обнаруживают, что число торговых агентов в месяце *N* (согласно спецификациям) есть функция числа покупателей в месяце *N*—1. Следовательно, при вычислении числа торговых агентов оказывается необходимым вычислять число клиентов. К сожалению, ситуация осложняется еще тем, что число клиентов есть функция числа торговых агентов и это, очевидно приводит к некоторой путанице!

Ситуация обычно усугубляется еще и тем, что один из программистов в это же время пытается выяснить, от чего зависит величина дохода гипотетического торгового предприятия в заданный месяц. Попытки представить эту величину в форме Фортран-выражения вида

ДОХОД=A + B + C + D

приводят к непреодолимым сложностям — в особенности по той причине, что почти каждый программист обнаруживает какое-то фантастическое стремление свести все вычисления к одному невообразимо сложному выражению! Это в свою очередь, вызывает жалобы слушателей: «Мы не можем решить эту задачу, так как не изучали в колледже дифференциальных уравнений». Те же, которым *довелось* изучать дифференциальные уравнения, видят в задаче не меньшие трудности; тут жалобы обычно выражаются в форме: «Я не могу решить эту задачу, так как не изучал в колледже бухгалтерского учета». Конечно же, колледж не дает универсального образования.

Интересно отметить, что некоторые группы программистов так и не приступили к росписи модуля ОСНОВНЫЕВЫЧИСЛЕНИЯ,

столкнувшись с трудностями при анализе модуля ОЦЕНКАПАРАМЕТРОВ, предложенного им в качестве элемента структуры нисходящего проектирования, отражаемой рис. 2.4а и 2.4б. И хотя, быть может, они понимали, *что* должно быть выполнено модулем ОЦЕНКАПАРАМЕТРОВ, они часто не понимали, *каким образом* это может быть сделано; другими словами, они не могли себе представить, как записать операции этого модуля. Не чувствуя в себе способности детально спроектировать модуль программы, они были озабочены тем, что проектирование всей программы ДЕНЬГИ кажется проблематичным, и часто пытались выйти из положения, оставив задачу проектирования и обращаясь к альтернативным средствам выполнения того, что должен был делать модуль ОЦЕНКАПАРАМЕТРОВ (иногда с помощью таких неуклюжих приемов программирования, как 48 вложенных один в другой циклов).

Эта озабоченность относительно возможности проектирования программы вполне понятна; удивляет лишь то, что они натолкнулись на такие трудности с модулем ОЦЕНКАПАРАМЕТРОВ, для реализации которого достаточно записать три или четыре оператора на Фортране или Коболе. Тем не менее эти трудности проектирования указывают на одно дополнительное обстоятельство, которое я нахожу еще более удивительным: в некоторых случаях я говорил программистам, чтобы они не беспокоились о реализации модуля ОЦЕНКАПАРАМЕТРОВ, что я напишу им эту программу сам. Их я просил лишь назвать, какие входные данные они посылают в мой модуль и какого характера выходы (результаты) желают получить. *Функция же,* которая должна быть реализована, может считаться известной (однако тот факт, что в действительности мы ее не представляем отчетливо, во многих случаях, как видно, явился причиной тех самых трудностей). Я столкнулся с тем, что в двух или трех случаях программисты отказались от такого предложения. Они пояснили, что до тех пор, пока они не поймут, что представляет собой модуль ОЦЕНКАПАРАМЕТРОВ во всех деталях, они не станут продолжать проектирование. И сколько бы раз я ни пытался убедить их в том, что этот модуль *может* быть запрограммирован и что я *сделаю* это для них, они упорно не желали продвигаться дальше. «Это просто не правильно,— утверждали они,— пытаться проектировать программу, содержащую столь загадочный модуль, что мы даже не знаем, как он работает».

Конечно, не все группы столкнулись с перечисленными выше трудностями; некоторым даже удалось найти хорошие решения, затратив на это приемлемое время. И тем не менее представляется интересным и поучительным осознать, как много оказалось трудностей, которые возникли в некоторых группах. Не в одной группе вынуждены были признаться, что после нескольких дней работы не было получено вообще никакого решения.

Таким образом, идея нисходящего проектирования в конце кон-

цов оказывается не такой уж простой. По-видимому, следует более пристально приглядеться к этому кажущемуся интуитивно очевидным принципу проектирования.

2.1.3. Более подробное рассмотрение нисходящего проектирования

Как мы уже видели, нисходящее проектирование включает разбиение большой задачи на меньшие подзадачи, которые могут рассматриваться порознь. Здесь, видимо, уместно сформулировать некоторые важные предпосылки для успешного применения нисходящего проектирования.

1. Представляется, что ключом к успешному нисходящему проектированию может служить формализованное и строгое описание проектировщиком *входов, функций* и *выходов* всех модулей программы или системы. Многие трудности, которые мне довелось наблюдать на практике проектирования программ, происходят по причине слишком неформального выполнения проектировщиками этой работы: «Ну, если нам требуется выполнить *X,* то мы просто запишем оператор CALL, и соответствующий модуль это нам сделает».

2. Как только вы убедитесь, что некоторая часть задачи *может* быть реализована в виде отдельного модуля, постарайтесь больше не думать об этом, т. е. не уделяйте слишком много внимания тому, как именно он будет реализован.

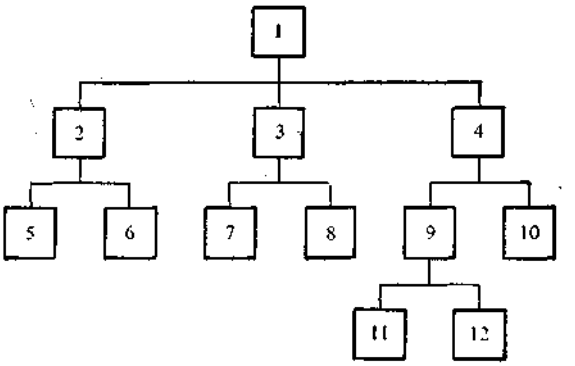
3. Внимательно следите за тем, чтобы вас (или группу проектировщиков) не вовлекли в обсуждение несущественных деталей задачи.

4. На каждом уровне проекта попытайтесь записать на одном листе реализацию модуля в виде последовательности программных кодов или соответствующей блок-схемы. Если для записи такой реализации модуля в терминах простейших элементов (например, с помощью простых операторов Кобола) одного листа недостаточно, то запишите ее в терминах модулей более низкого уровня, которые должны быть спроектированы на одной из последующих стадий.

5. Проектированию структуры данных следует уделять не меньше внимания, чем проектированию процессов или алгоритмов. Во многих случаях в состав этих данных входят требования к межмодульным *интерфейсам,* и проектирование этих модулей не продвинется существенно до тех пор, пока не будут тщательно описаны соответствующие интерфейсы.

Более подробно некоторые из этих предпосылок обсуждаются ниже.

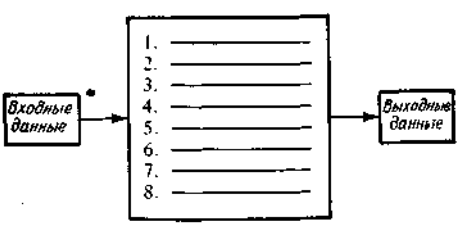
Спецификация интерфейсов. Ранее мы отметили, что одним из наиболее важных элементов нисходящего проектирования является формализованный подход к спецификации входов, функций и выходов, которые должны быть реализованы каждым модулем. Здесь



СОДЕРЖАНИЕ

1.	7.
2.	8.
3.	9.
4.	10.
5.	11.
6.	12.

а



б

Рис. 2.5. Диаграммы HIPO.

а — общий вид; *б* — подробные диаграммы. Каждый пункт в такой диаграмме может ссылаться на диаграмму более низкого уровня.

уместно привести разочарованные признания одного из обучавшихся новому подходу относительно его попыток нисходящего проектирования программ на Коболе. «Это был мой первый опыт использования подпрограмм, и я представлял себе это так, что если я придумал какой-нибудь модуль, к которому можно обратиться из главной программы оператором PERFORM, то все остальное устроится само собой».

Ничто, однако.не устраивается само собой. Напротив, поверхностное восприятие нисходящего проектирования рождает у прог-

раммиста ложное чувство самоуверенности.Именнопо этойпричине в некоторых организациях были предприняты энергичные попытки *формализовать* этапы нисходящего проектирования. Интересно то, что многие из этих попыток нашли свое отражение в руководствах по проектированию, используемых в области автоматизированной обработки данных, как средство организации и управления комплексной разработкой проектов соответствующих систем. Одним из наиболее интересных и широко распространенных приемов такого формализованного подхода к нисходящему проектированию является созданная на фирме IBM методология иерархических диаграмм вход — обработка — выход или сокращенно HIPO. Согласно этой методологии, полная структура системы должна быть представлена с помощью диаграммы вида, изображенного на рис. *2.5а*. Каждый модуль этой диаграммы в свою очередь представляется посредством диаграммы вида, изображенного на рис. 2.5*б*. При необходимости рисуютподробныедиаграммымодулейболеенизкихуровней. Еще более формализованная модель графического представления структуры систем предложена в недавно вышедшей статье Стивенса, Майерса и Констентайна [6]. Не следует, однако, думать, что HIPO является единственно возможной методологией или что графическое представление Констентайна — единственно возможный подход. Главное здесь состоит в том, что в нисходящем проектировании следует использовать некоторый формализованный и регламентированный подход к документированию; в противном случае программисты и проектировщики обнаруживают тенденцию к слишком неформальному описанию.

Важно отметить недостаточность в этом смысле различных языков программирования. Программист, работающий на Коболе, часто представляет себе интерфейс между модулями программы в терминах глагола PERFORM, хотя этот оператор не требует какого-либо явного задания перечня аргументов. Некоторые могут подумать, что программирование на Фортране предполагает явное задание формальных параметров, посылаемых вызываемой подпрограмме, поскольку правило вызова подпрограмм задается следующей общей формой

```
CALL GLOP (A, B, C)
```

и тем не менее интересно отметить, насколько часто программисты на Фортране пользуются для передачи аргументов оператором COMMON только потому, что им лень выделить эти аргументы в отдельный список формальных параметров. Эти замечания, очевидно, применимы и к ПЛ/1 и другим языкам высокого уровня.

Некоторые программисты жалуются, что процесс проектирования программы или вычислительной системы *нельзя* описать столь формально, что он представляет последовательность интуитивных попыток, в которых формируется решение. Они часто отмечают, что

это итеративный процесс и что трудности, возникающие на нижнем уровне проекта, часто вынуждают переосмысливать решения, принятые на более высоком уровне. Безусловно, некоторые этапы процессов проектирования *относятся* к этому типу; на самом деле никто и не мыслит себе создание законченного проекта как процесс, напоминающий единый порыв блестящего гения. В особенности, если перед нами случай, когда проектирование выполняется группой людей, то мы должны быть готовы к тактике «два шага вперед, шаг назад»; нам следует допускать, что проектировщики могут изменить свое мнение о ранее выбранной реализации некоторого модуля; мы можем согласиться с тем, что первая попытка создания проекта программы выглядит несколько расплывчатой и неопределенной. Это ни в коей мере не нарушает основных принципов нисходящего проектирования. После «мозгового штурма» и достаточно полного неформального обсуждения деталей проекта определенного уровня программы все-таки следует потребовать, чтобы эта часть проекта была записана формально (в терминах входов, функций и выходов), прежде чем мы перейдем к проектированию следующего уровня. Если трудности, возникшие при проектировании уровня номер $N+1$, действительно вызывают необходимость повторного проектирования уровня номер N, то это и следует сделать; и тем не менее нам представляется, что такой подход к решению задачи остается регулярным.

Старайтесь не обращать внимания на детали описания модулей нижних уровней. Как было отмечено выше, многие программисты увлекаются описанием несущественных деталей проекта, в то время как требуется сосредоточить внимание на более высоких его уровнях. Здесь, по-видимому, уместно спросить: «В чем причина этого?» и «Что в связи с этим можно предпринять?». Наблюдая несколько групп программистов, попавших в эту ловушку, и спрашивая, *что* ввергло их в обсуждение мелочей, я понял, что тут могут быть выделены три основные причины.

1. Некоторые программисты признают, что они могут обсуждать как единственно осязаемые вещи только мелкие подробности. Большую часть построений, касающихся верхних уровней проекта, они находят слишком абстрактными и неопределенными. Поскольку в своем большинстве они не имеют достаточного опыта в проектировании такого рода (в действительности возникает вопрос, имеют ли они вообще *какой бы то ни было* опыт в проектировании программ?), то им трудно общаться друг с другом. И все они соглашаются с тем, что по крайней мере конкретные детали проекта могут служить для них предметом обсуждения.

2. Многие группы программистов подтверждают, что основная проблема для них заключается в дисциплине. Даже понимая, что они напрасно тратят время на обсуждение тривиальных аспектов проекта, они обнаруживали, что их внимание неизменно оказыва-

ется привлеченным именно к этим аспектам. Во многих случаях это отражение трудностей управления; в группе не нашлось достаточно авторитетного человека, который настоял бы на том, чтобы обсуждение несущественных предметов было отложено.

3. Наконец,внекоторыхгруппахзаявили,чтоуспех всегопроекта может зависеть от того, смогут ли они найти удовлетворительное решение для определенного модуля нижнего уровня. Например, мне довелось слышать от проектировщиков программ, работающих в реальном масштабе времени, следующее объяснение. «Если я не могу добиться того, чтобы мой модуль *X* нижнего уровня выполнял свои операции за 43 мкс, то вся система окажется неудовлетворительной». Хотя в отдельных случаях это может быть и так, обычно эти опасения преувеличены как бы то ни было *после того,* как критический модуль спроектирован, нет никаких причин, по которым проектировщик не мог бы вернуться к методологии нисходящего проектирования. Тревожит то, что многие проектировщики программ, углубившись в проектирование модуля нижнего уровня, *забывают* вернуться к целостному проектированию программы или системы.

Каким образом можно преодолеть эту трудность? Вомногихслучаях достаточно повторить предостережение предыдущего раздела: заставьте себя тщательно и формализованно описать интерфейс между верхними уровнями программы и тем модулем нижнего уровня, необходимость которого вы только что осознали и реализацией которого так озабочены. В большинстве случаев сразу должно становиться ясным, что этот модуль нижнего уровня *можно* реализовать; очень может быть, что именно нечеткость определения модуля прежде всего и оказывается причиной указанных трудностей. Даже в том случае, когда реализация такого модуля не вполне очевидна, здравый смысл должен вам подсказать, может ли он быть составлен кем-нибудь, кроме суперпрограммиста. Если реализация кажется очень трудной или имеет очень важное значение, как в приведенном выше примере, касающемся систем реального времени, то было бы разумным заняться его осуществлением. Важно при этом, опять же, чтобы нисходящий процесс был возобновлен как только оказываются разрешенными все трудности в задаче нижнего уровня.

Ограничивайте размеры модулей в нисходящем проектировании. Проектируя, вы должнытакжепомнитьхорошееправило: страницадве текста за один прием. На каждой стадии проектирования программы вам надо пытаться записать в явном виде реализацию некоторого модуля, для которого определены входы, функции и выходы. Если реализация модуля в виде тридцати-сорока команд (например, Кобол-, Фортран- или ПЛ/1-операторов) не представляется очевидной, то попытайтесь определить важнейшие *вспомогательные функции* модуля.

Хотя порой это сделать не очень просто, однако почти всегда возможно представить данный модуль в виде относительно небольшого числа подмодулей. Сделав это несколько неформально (с тем, чтобы просто упорядочить характер мышления), вы можете приступить к реализации каждого из подмодулей, стремясь представить его в виде последовательности простейших операторов и обращений к модулям более низкого уровня.

Допустим, например, что перед нами стоит задача спроектировать типичную программу. Вместо того чтобы погрузиться в анализ огромного числа составляющих элементов, мы могли бы организовать наши представления следующим образом.

```
CALL INITIALIZATION
CALL READINPUT
CALL EDITINPUT
CALL PROCESSINPUT
CALL TERMINATION
```

Хотя на этом уровне проектирования результат может выглядеть тривиальным, тем не менее это полезно для организации наших мыслей, подобно тому как бывает полезным перечислить основные вопросы, прежде чем пытаться составлять отчет или памятную записку. Взглянув на эту форму программы более критически, мы могли бы даже переписать ее в виде

```
CALL INITIALIZATION
CALL READINPUT
CALL EDITINPUT
IF FATALERRORFLAG = 0 THEN
     CALL PROCESSINPUT
ELSE
     CALL PRINTERRORMESSAGE
CALL TERMINATION
```

К этому моменту нам могло бы стать ясным, что начальные операции довольно просты, например они могли бы сводиться к открытию двух файлов и заданию начального значения переменной FATALERRORFLAG. Аналогично модуль заключительных действий может быть столь же простым, включающим лишь закрытие тех же файлов. Таким образом, мы могли бы найти приемлемой запись проекта нашей программы в следующей форме:

```
OPEN FILEA
OPEN FILEB
FATALERRORFLAG = 0
CALLREADINPUT
CALL EDITINPUT
IF FATALERRORFLAG = 0 THEN
     CALL PROCESSINPUT
ELSE
     CALL PRINTERRORMESSAGE
CLOSE FILEA
CLOSE FILEB
```

Заметим, что в приведенном примере оказалось нарушенным, и весьма основательно, ранее высказанное требование формального определения входов, функций и выходов каждого модуля. Мы, например, не определили точно ни функций модуля READINPUT, ни *его* входов, ни его выходов. Тем не менее этот абстрактный пример показывает, что мы можем представить общий вид программы, употребив около десяти простейших операторов и обращений к подпрограммам более низкого уровня описания.

Тщательно проектируйте структуру данных. Наконец, следует подчеркнуть, что обычно в проектировании описание структуры *данных* имеет не менее важное значение, чем описание алгоритмов. В большинстве случаев идеология нисходящего проектирования применима к описанию данных в той же мере, в какой она применима к описанию алгоритмов, хотя и возможны значительные отличия одного проекта от другого. Так, например, некоторыми руководствами по организации проекта рекомендуется применение нисходящего проектирования системных модулей. При этом предполагается, что вначале разработчик системы описывает *файлы,* которые будут использованы системой, после чего в качестве определенного этапа проектирования должна следовать идентификация записей, содержащихся в этих файлах, и, наконец, детальная спецификация каждого формата каждого поля в записях.

Некоторые разработчики систем утверждают, что в основном структура базы данных известна заранее. Для них, например, представляется очевидным, что их прикладные экономические программы обращаются к главному файлу, файлу изменений и некоторым другим типам файлов. В том, что их *касается,* остается лишь определить соотношения между одними записями и другими (например, правило упорядочения записей файла), элементы содержания каждой записи и т. д. Тот факт, что структура файлов кажется очевидной, аналогичен ситуации, в которой многие разработчики систем обнаруживают, что структура их программы может быть представлена в стандартной форме, показанной на рис. 2.1 или 2.2.

Однако в более общем случае мы должны считать, что база данных, о которой здесь идет речь, включает входные данные всей программы, ее выходные данные и различные файлы, таблицы и другие вспомогательные переменные, которые передаются в программе из одного модуля в другой. Обычно мы можем считать, что выходные результаты точно определены заранее. Во всяком случае, это входит в основные обязанности специалиста по системному анализу, который работает с заказчиком и будущим пользователем программы. Если он выполнил работу как следует, то основные функции программы также оказываются точно определенными заранее. Часто нам самим приходится объяснять пользователю, в каком виде должны задаваться значения входных данных для того, чтобы наша про-

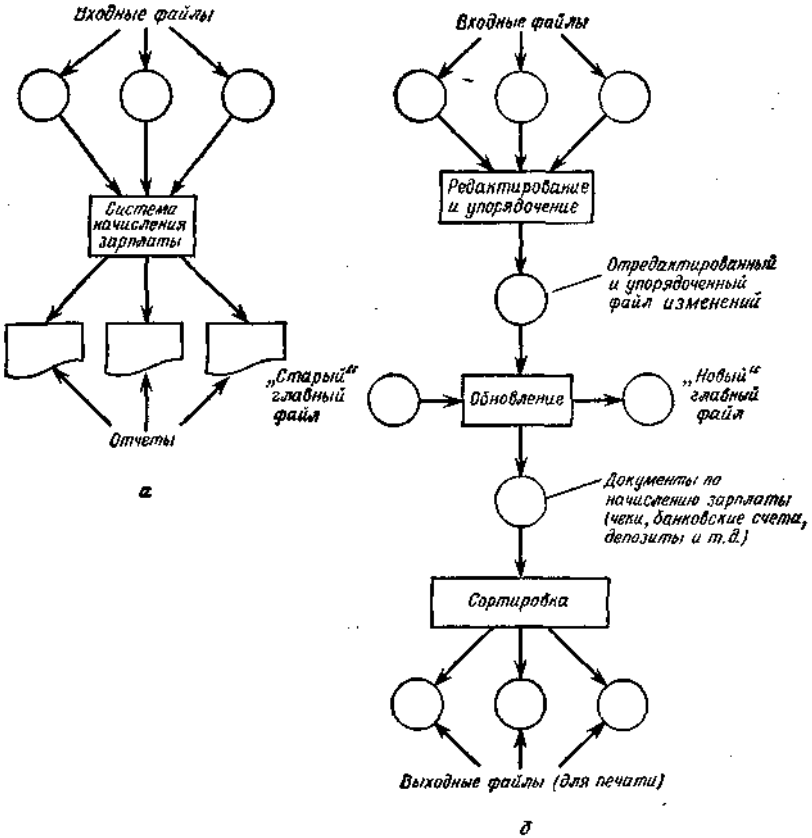


Рис. 2.6. Система начисления заработной платы.

а — структура верхнего уровня гипотетической системы начисления заработной платы; *б* — более подробное представление системы начисления заработной платы.

грамма могла выдать значения выходных данных в той форме, в которой он желает их видеть. Во всяком случае, обычно, мы можем выбрать по своему усмотрению форматы внутренних файлов, таблиц и других переменных.

Может оказаться, что для успешного нисходящего проектирования программных модулей некоторые элементы базы данных должны быть определены до *мельчайших* подробностей на относительно ранних этапах проектирования. Рассмотрим, например, программу начисления заработной платы, общая структура которой представлена на рис. 2.6, *а*. На этом этапе мы можем считать, что формы как входных, так и выходных данных уже точно определены и фиксиро-

ваны, в противном случае мы не могли бы быть уверены, что форма выходных данных удовлетворит пользователя или что он смог бы обеспечить необходимую форму входных данных. Более того, если бы мы захотели расширить представление нашей гипотетической системы до вида, показанного на рис. 2.6,6, прежде чем сделать следующий шаг в проектировании редактирующего и обновляющего модулей, мы должны были бы предположить, что все непосредственно используемые файлы точно определены вплоть до последнего бита информации в каждой записи.

Важно заметить, что мы делаем различие между *подробным* и *негибким* проектированием структуры данных. Основное назначение нисходящего проектирования — служить средством разбиения большой задачи на меньшие подзадачи таким образом, *чтобы каждую подзадачу можно было рассматривать независимо*. Так, для того чтобы редактирующий и обновляющий модули нашей гипотетической системы начисления зарплаты могли быть спроектированы порознь, мы должны иметь настолько детализированное описание соответствующих интерфейсов, что (в теории) у разработчиков редактирующего и обновляющего модулей не было бы необходимости общаться друг с другом в процессе проектирования. Это, однако, не освобождает нас от условия гибкого проектирования названных интерфейсов; так, мы могли бы потребовать от разработчиков обоих модулей, чтобы они предусмотрели возможность введения новых видов налогов, новых принципов начисления зарплаты (например, вместо еженедельного — поденного) и т. д.

2.1.4. Разработка структурированных спецификаций к проекту

Обсудив на нескольких последних страницах проблемы нисходящего проектирования, в заключение мы должны назвать наиболее сложную из них: *чрезвычайно трудно осуществлять упорядоченное нисходящее проектирование, исходя из несогласованных, неполных и беспорядочно составленных требований.* К сожалению, оказывается, что именно такого характера спецификации мы чаще всего и получаем!

От кого обычно мы получаем требования на разработку программы или системы? Если они получены от пользователя, не являющегося специалистом в области вычислительных наук, то у нас нет оснований ожидать от него грамотно составленных спецификаций; это становится частью нашей работы как профессиональных программистов-разработчиков. В то же время, если мы получаем спецификации от специалиста по системному анализу, который работает с пользователем и обязан знать, что именно ему нужно, то мы *вправе* ожидать что-то более упорядоченное. В действитель-

ности мы могли бы требовать, чтобы сами требования были организованы в соответствии с идеологией нисходящего проектирования.

Ясно, что это может оказаться трудной задачей. Не имея намерения нанести обиду множеству специалистов по системному анализу во всем мире, мы должны тем не менее указать на следующие возможные ситуации:

1. Некоторые подвизаются в области системного анализа только в результате действия известного принципа Питера: они были блестящими программистами и с повышением получили должность, связанную с системным анализом, в котором они некомпетентны.

2. Некоторые аналитики систем не были даже хорошими программистами, во всяком случае по современным представлениям. Может быть, они могли заставить машину IBM 1401 встать на голову и выплевывать деревянные 5-центовики; но им, видимо, *не* довелось изучить (или натолкнуться на) важные современные концепции нисходящего проектирования, модульного проектирования, структурного программирования и т. д. По этой причине спецификации таких аналитиков часто являются отражением их прежнего опыта беспорядочного программирования.

3. Некоторые аналитики систем вообще *не* имели дела с программированием, например они могли перейти в область системного анализа из среды пользователей. В результате они совершенно незнакомы с понятиями из области проектирования программ, которые здесь рассмотрены; может случиться, что они не знакомы вообще с какими-либо формами логического мышления.

Можно, естественно, ожидать, что системные аналитики не менее резко отзываются о программистах, и мы, возможно, нашли бы подобные высказывания в какой-нибудь книге по системному анализу! Тем не менее представляется, что успех в применении программистом (или разработчиком системы) нисходящего проектирования может сильно зависеть от характера спецификаций, с которыми он работает.

Первое, что здесь может быть предложено, состоит, конечно, в том, что аналитики систем (а в некоторых случаях, возможно, и пользователи) должны быть осведомлены относительно некоторых понятий и принципов нисходящего проектирования. Это еще более важно в вопросах нисходящего тестирования, обсуждаемых в разд. 2.3. Естественно, что это предполагает знакомство системного аналитика с основами аппаратных средств и программирования для ЭВМ,— предположение, которое, кажется, вызывает энергичное несогласие многих аналитиков систем! Во-вторых, аналитик должен помогать программисту, используя при составлении спецификаций нисходящую их организацию; на самом деле большая часть того, что здесь рассматривается в связи с нисходящим проектированием, применима и к методологии разработки требований по нисходящей схеме.

Довольно энергично высказывались мнения, что требования следует составлять, употребляя некоторую структурированную форму записи, в противоположность обычному употреблению для этой цели неоднозначного естественного языка. Это привело даже к созданию «структурного английского языка» — способу записи, включающему лишь императивные конструкции, предложения типа IF-THEN-ELSE и предложения в форме DO-WHILE, эта точка зрения согласуется с идеями структурного программирования, подробно обсуждаемыми в гл. 4. В то же время другими разработчиками систем высказывались требования, чтобы спецификации оформлялись в виде таблиц решений или в соответствии с методологией HIPO, которую мы рассмотрели выше в этом разделе.

2.2. Нисходящее кодирование

Подробно обсудив нисходящее проектирование, мы теперь обратимся к другому тесно связанному с ним предмету — нисходящему кодированию, или непосредственномунаписанию программ. Сначала мы определим его основные принципы, потом приведем некоторые доводы в пользу их применения и затем обсудим некоторые детали процесса кодирования.

2.2.1. Определения

Нисходящее кодирование базируется главным образом на представлении о параллельности различных этапов проектирования программы и кодирования отдельных ее частей. Насколько можно судить, в литературе пока нет соответствующей установившейся терминологии. Некоторые употребляют слова «нисходящее программирование» для обозначения того, что мы называем нисходящим проектированием; другие пользуются словами «нисходящее программирование» для описания совокупных представлений «нисходящего проектирования», «нисходящего кодирования» и «нисходящего тестирования». Чтобы избежать недоразумений, оговоримся, что мы будем употреблять слова «нисходящее кодирование» для обозначения процессов написания кодов программы; нам представляется интересным выяснить, *когда* следует писать эти коды.

В простейшей форме нисходящего кодирования подразумевается, что к написанию кодов приступают лишь после того, как полностью закончено проектирование программы. Завершив проектирование, мы прежде всего пишем коды главной программы; когда эта работа закончена, мы кодируем модули более низкого уровня и т. д. Хотя эта идея может показаться довольно тривиальной, в ее пользу можно привести некоторые разумные доводы, как это будет видно из следующего раздела.

В большинстве случаев, обсуждая нисходящее кодирование, имеют в виду нечто более претенциозное. В крайней форме этих представлений заключается следующее: по завершении проектирования верхнего уровня программы и *до начала проектирования следующих уровней, должны быть написаны коды этого верхнего уровня*. Затем проектируется и кодируется второй уровень, после этого — третий и т. д. В этой методологии, очевидно, допускается, что некоторое время мы заняты кодированием при незавершенном проектировании.

Как можно ожидать, эта крайняя форма нисходящего кодирования наводит страх на многих программистов; специалисты большинства организаций, занятых автоматизированной обработкой данных, ознакомившись с этой методологией, находят ее настолько радикальной, что обнаруживают крайнее нежелание даже опробовать ее. Тем не менее, как мы увидим в следующем разделе, существуют некоторые важные доводы в пользу этой методологии.

2.2.2. Доводы в пользу нисходящего кодирования

Доводы в пользу нисходящего кодирования можно свести к следующему:

1. Блок-схемы, структурные диаграммы и другие методы часто оказываются недостаточными средствами описания проекта. Во многих случаях программный код оказывается более точным, адекватным и удобным.

2. В процессе кодирования могут быть вскрыты те или иные трудности проектирования логики более низких уровней программы просто потому, что обычно при кодировании программист вынужден более тщательно продумывать свои действия. Было бы неплохо осознать существование таких трудностей как можно раньше, пока они еще могут быть легко устранены.

3. Нисходящее кодирование упрощает нисходящее тестирование, последнее является предметом обсуждения разд. 2.3.

Даже если бы два первых довода были бы ложными или несущественными, для применения нисходящего кодирования было бы достаточно того, что на нем основывается нисходящее тестирование; обычно это *важнейший* довод в пользу нисходящего кодирования. Как мы рассмотрим это более подробно в разд. 2.3, идея нисходящего тестирования предполагает, что мы приступаем к тестированию программы еще до того, как завершено ее проектирование. Это позволяет нам раньше опробовать основные межмодульные интерфейсы, а также убедиться в том, что программа в основном удовлетворяет требованиям пользователя. Чтобы осуществить такое тестирование, необходимо написать коды более высокого уровня проекта программы прежде, чем будет завершено проектирование более низких уровней,

Другие два из упомянутых доводов считаются многими программистами не менее важными. Как можно заметить, в большинстве случаев рассмотрение процесса нисходящего проектирования сопровождается использованием реального кода (или, какего называют, «Эсперанто ЭВМ») как средства описания проекта. В то же время некоторые программисты настаивают на том, что *графические* формы представления проекта, такие, как блок-схемы и структурные диаграммы, вызывают у тех, кому их показывают, значительно большее впечатление. Тот факт, что блок-схемы часто содержат недостаточную информацию, не может, по-видимому, служить доводом для отказа от графической формы представления. Возможно, нам нужен более структурированный подход к графическому представлению программ, какой, например, выбран Констентайном и др. в работе [6].

Можно привести некоторые доводы в пользу блок-схем как средства *обмена идеями,* но в то же время многие программисты признают, что блок-схемы как способ *документирования* структуры программы часто оказываются неудовлетворительными по той простой причине,что блок-схемам редко уделяютдостаточновнимання после того, как программа разработана. Большинство программистов согласятся с тем, что они редко обременяют себя составлением блок-схемы, если программа уже написана (да и то лишь по настоянию руководителя), так что нет гарантии в том, что программа и блок-схема находятся во взаимно-однозначном соответствии. В то же время очевидно, что код программы всегда отвечает ее текущему состоянию в процессе ее естественного сопровождения и развития.

Таким образом, мы придерживаемся того мнения, что код программы может служить одним из лучших способов передачи информации по проекту от одного человека к другому и что в большинстве случаев это и лучшая форма документирования проекта. Простота кодовых записей как средства общения особенно очевидна при описании верхних уровней проекта. То, что программист обыкновенно описывает с помощью блок-схемы, изображенной на рис. 2.7а, столь же просто может быть записано в форме кодов, показанных на рис. 2.7б. Часто это справедливо и в отношении более низких уровней логической структуры программы, *в особенности, если программист следует правилу: никогда не проектировать фрагментов, требующих для своей записи более одной страницы.*

По мнению многих программистов, в том числе и автора, блок-схемы часто являются хорошим инструментом упорядочения мысленных образов, в особенности на начальном этапе проектирования программы, когда большую часть блестящих идей программист записывает на оборотной стороне почтового конверта или на какомнибудь подвернувшемся под руку клочке бумаги. Однако, организовав свои образы для себя, мы должны представить их для восприя-

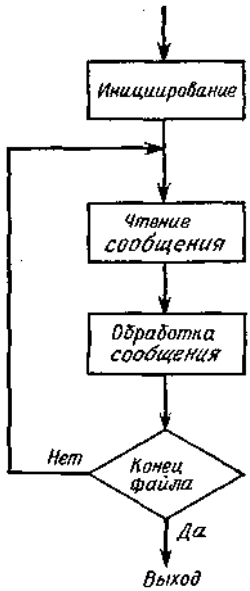


Рис. 2.7а. Блок-схема верхнего уровня.

тия другими людьми в форме скрупулезно составленных и аккуратно записанных кодов — точно так, как в установившейся практике программисты представляют для своих сообщений проекты программ в форме скрупулезно составленных и аккуратно нарисованных блок-схем. Имеются основательные причины для предположений, что будущие программисты смогут обходиться вообще без блок-схем: их первым действием могла бы служить запись нескольких пробных строк кода программы. Во всяком случае, неплохо было бы различать «личное», или «частное», проектирование — наметки проекта программы, записываемые на обороте конверта, от «общественного». Какими приемами пользуется программист при «личном» проектировании, может быть известно емусамому да господу богу; совсем другое — в какой форме он представляет результаты проектирования той или иной аудитории.

```
        PERFORM INITIALIZATION.
        PERFORM MAIN-LOOP UNTIL
             END-OF-FILE-FLAG= I.
        STOP RUN.
MAIN-LOOP.
        READ TRANSACTION-FILE AT END
             MOVE 1 TO END-OF-FILE-FLAG.
        PERFORM PROCESS-TRANSACTION.
        EXIT.
```

Рис. 2.7б. Программный код к блок-схеме, показанной на рис. 2.7*а*.

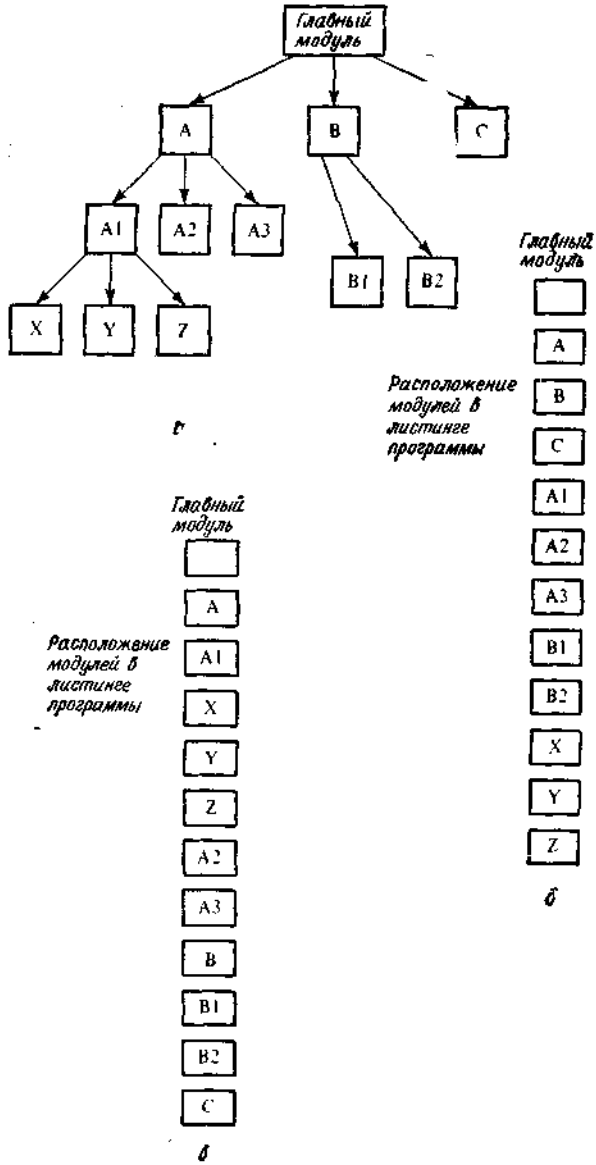


Рис. 2.8.
а — типичная программа; *б* — горизонтальная организация расположения модулей; *в* — вертикальная организация модулей.

Существует еще один аспект нисходящего кодирования, который заслуживает краткого обсуждения. Если предположить, что мы уже *имеем проект* программы, организованный в соответствии с идеями нисходящего проектирования, то каким образом мы должны расположить в листинге программы коды, отвечающие различным ее модулям? Допустим, например, что мы спроектировали программу, изображенную на рис. 2.8,а; мы могли бы рассматривать такие варианты организации кода программы, как горизонтальная, показанная на рис. 2.8,б, или вертикальная, представленная рис. 2.8,в. Если мы хотим отобразить общий вид программы, горизонтальная организация модулей представляется более удобной: взглянув на модуль главной программы, мы, скорее всего, захотели бы рассмотреть коды модулей Л, *В* и С, которые в этой организации листинга программы располагаются один под другим. Аналогично, изучив модуль В, нам, по всей вероятности, было бы интересно увидеть представления модулей В1, В2, В3, которые опять размещены рядом.

С точки зрения задач отладки или сопровождения вертикальная организация листинга оказывается более удобной. Если мы прослеживаем логику программы, мы, скорее всего, захотим изучить главную программу, затем коды модуля А, затем коды модуля А1, затем *X, Y,* Z, затем А2 и так далее. Если мы организовали модули, как это показано на рис. 2.8,в[1)], то соответствующие коды в листинге встречаются в желаемом порядке.

Конечно, основная проблема, с которой мы здесь сталкиваемся, состоит в следующем: если мы рассматриваем некоторый, наугад выбранный из середины листинга нашей программы, модуль X и видим, что он ссылается на модуль *Y,* то как узнать, где располагается модуль *Y?* Используя, например, горизонтальную организацию программы и исследуя модуль В, каким образом мы можем определить расположение модуля В1 в листинге программы? Аналогично, следуя вертикальной организации программы, при рассмотрении модуля А, как нам быстро найти модуль А3? Были предприняты поиски различных способов нумерации модулей или их лексикографического упорядочения (как в самом примере, рис. 2.8, *а),* и оказалось, чтоэти способы обычно бывают приемлемыми, если употребить немного здравого смысла. Например, если известно, что программист применил вертикальную организацию своей программы, то мы знаем, что все подмодули модуля А появляются в листинге программы раньше любого из подмодулей модуля В. Обычно мы не возражаем против необходимости поиска среди подмодулей данного модуля А того подмодуля, который нам нужен. Естественно, что этот процесс упрощается, если программист

[1)] При анализе структуры данных эту схему упорядочения обычно называют «предупорядочением» (см. Кнут, Искусство программирования для ЭВМ, т. 1, изд-во «Мир», М., 1976).

придерживается некоторого правила в обозначениях и организации его модулей, и тем более это так, если столь же последовательными оказываются все программисты, работающие над одним и тем же проектом.

2.3. Нисходящее тестирование

Следующей важной концепцией, которую следует рассмотреть, является концепция *нисходящего тестирования*. Как и в предыдущем разделе, вначале мы дадим описание нисходящего тестирования общего характера, а затем обсудим его достоинства; в конце мы рассмотрим некоторые модификации нисходящего тестирования, применявшиеся на практике в некоторых программных разработках.

2.3.1. Основные понятия нисходящего тестирования

Классический метод тестирования, именуемый в наше время словами «восходящее тестирование» (или «тестирование снизу вверх»), иллюстрируется рис. 2.9, *а — в*. Первый этап тестирования обычно

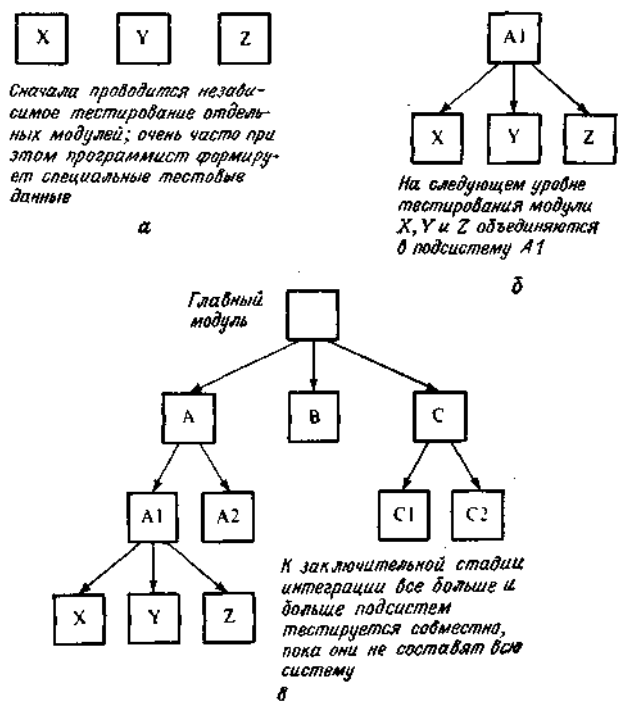


Рис. 2.9. Восходящее тестирование.

определяют как «тестирование модулей», «тестирование кодов» или «тестирование на уровне элементов»; следующий этап «пусковые испытания» или «тестирование подсистем»; заключительный этап может быть определен словами «испытания в системе», «интеграция» или «эксплуатационные испытания». Эта методология, по-видимому, принята в качестве стандартного подхода к тестированию во многих организациях и была рекомендована в ряде учебников по программированию и руководств по управлению проектированием.

В качестве альтернативы распространенной методологии восходящего тестирования некоторыми специалистами в программировании для ЭВМ в настоящее время предлагается подход, в котором тестирование организуется по нисходящей схеме в согласии с методологией нисходящего проектирования и нисходящего кодирования, рассмотренными в этой главе выше. Основная идея этого подхода довольно проста. Как это показано на рис. 2.10,а, сначала мы тестируем модуль главной программы и модули одного или двух более низких уровней. После того как это логическое ядро программы испытано настолько, что у нас появляется уверенность в правильности реализации основных интерфейсов, мы присоединяем следующие уровни логической структуры программы (рис. 2.10, *б).* Когда система пополняется самым нижним уровнем модулей (рис. 2.10, б), наша задача оказывается эффективно решенной, т. е. на этой стадии не требуется тестирования всей системы.

Методология нисходящего тестирования предполагает использование фиктивных модулей, или фиктивных программ. Как показано на рис. 2.10, *а,* к рассматриваемому моменту оказались реализованными лишь несколько верхних уровней проекта программы; нижние уровни пока отсутствуют. Это означает, что модули верхних уровней обращаются с помощью операторов CALL или PERFORM к модулям более низких уровней, причем эти модули должны быть представлены одной из таких временных реализаций, как:

1. Прямая передача управления на выход из модуля, если выполняемая им функция не является критической.
2. Выдача постоянного значения выходных данных.
3. Выдача случайного значения выходных данных.
4. Печать отладочного сообщения, указывающего на то, что управление было передано на вход модуля.
5. Реализация упрощенного варианта окончательной версии модуля. Допустим, например, что в одном из модулей нижнего уровня необходимо вычислить квадратный корень из числа; окончательная версия модуля может быть написана на языке ассемблера с применением алгоритма, минимизирующего число требуемых арифметических операций, но в качестве «простейшего» варианта может быть использована довольно неизящная Фортран-программа вычисления корня, основанного на методе Герона.

Теоретически мы могли бы начать нисходящее тестирование,

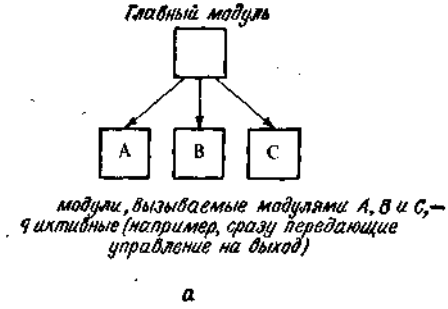


Рис. 2.10. Нисходящеетестировани.

имея реализацию лишь главной программы с фиктивными модулями нижних уровней; на практике, конечно, это выглядело бы довольно нелепо. Здравый смысл подсказывает нам, что число уровней, реализованных в виде программных модулей, должно быть достаточным, чтобы образовать логический скелет программы; последующие этапы тестирования можно представлять себе как наполнение этого скелета плотью кодов, пока программа не обретет законченный вид.

Должно быть ясно, что нисходящее тестирование используется в согласии и параллельно с нисходящим проектированием и нисходящим кодированием. В идеальном случае предполагается, что мы должны спроектировать главную программу, написать ее коды и выполнить тестирование; затем спроектировать следующий уровень модулей, написать коды соответствующих программ и ввести их в существующий логический скелет программы для проведения тестирования; мы должны продолжать таким образом до тех пор, пока не будут спроектированы, закодированы и введены в програм-

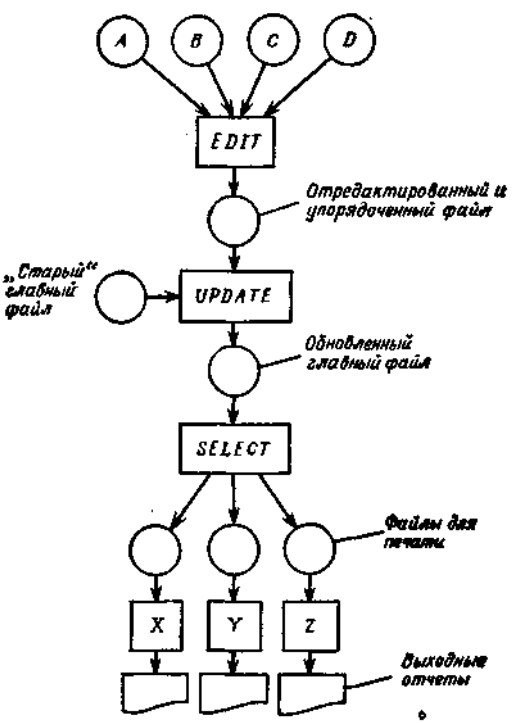


Рис. 2.11. Пример нисходящего тестирования,

му модули самого нижнего уровня. Как мы увидим в разд. 2.3.3, существуют различные модификации этой общей схемы, которые могут быть использованы в том или ином конкретном проекте.

Естественно, что осуществить *полное* тестирование программы невозможно, пока она представлена лишь скелетной схемой, изображенной на рис. 2.10, *а.* Однако мы можем опробовать интерфейсы между основными модулями программы и убедиться в том, что выходные данные одного модуля действительно могут служить входными данными для другого. Добавив следующий уровень модулей, как это показано на рис. 2.10, *б,* мы смогли бы провести значительно более полное тестирование, поскольку каждый основной модуль программы точнее выполнял бы функции, которые связываются с его окончательной реализацией. На каждом уровне, или этапе, тестирования наша цель состоит в том, чтобы, наблюдая прохождение некоторых данных через всю программу, убеждаться в правильности совместного функционирования всех интерфейсов.

На рис. 2.11 приведен пример, иллюстрирующий методологию нисходящего тестирования. Как видно, модуль РЕДАКТИРОВАТЬ должен быть приспособлен к приему четырех различных файлов сообщений; его назначение состоит в анализе возможных ошибок в сообщениях и упорядочении их в наперед заданной последовательности. Выходные данные модуля РЕДАКТИРОВАТЬ (организованные в виде файла, хотя это и не обязательно) являются входными данными для модуля ОБНОВИТЬ; модуль ОБНОВИТЬ объединяет сообщения, сформированные модулем РЕДАКТИРОВАТЬ, с записями главного файла и формирует новое состояние главного файла. В свою очередь обновленный главный файл используется в качестве входного другим модулем программы, называемым ВЫБРАТЬ. Как подсказывает его название, модуль ВЫБРАТЬ извлекает различные записи из главного файла, упорядочивает их (возможно, различными способами) и формирует три выходных файла. Программа ВЫБРАТЬ может, например, извлечь из главного файла все записи типа X и расположить их в алфавитном порядке. Аналогично он может извлечь все записи типа Y и упорядочить их по номерам, используя один из ключей записи, записи типа Z могут быть извлечены и упорядочены с использованием другого ключа записи. Наконец, некоторые программы формирования отчетов считывают выходные данные модуля ВЫБРАТЬ и выдают на печать соответствующие отчеты.

Не зная ничего более о характере процессов, выполняемых каждым из главных модулей, мы уже на этом этапе можем начать планирование программы нисходящего тестирования. Каждый программист может выбрать в какой-то мере индивидуальный подход, и, конечно, конкретную последовательность тестовых операций следует определять по здравому смыслу. В качестве примера тем не менее мы могли бы предложить следующие семь шагов.

1. Должны быть подготовлены управляющие карты для всех модулей или всех программ. Вся программа должна выполняться с фиктивными файлами и модулями, т. е. программа РЕДАКТИРОВАТЬ, так же как и программы ОБНОВИТЬ, ВЫБРАТЬ и программа печати, должна сразу по получении управления передавать его на выход. Хотя этот шаг кажется тривиальным, правильное формирование управляющих карт может представлять одну йз основных трудностей, например, при работе с ЭВМ Систем IBM/360 и IBM/370. Тем, кто утверждает, что это — тривиальная операция, можно ответить: если она тривиальна, то вы справитесь с ней за несколько минут, и, поскольку рано или поздно нам придется это сделать, почему бы не написать управляющие карты сейчас, с тем чтобы иметь некоторый логический скелет для следующего шага?

2. Цель следующего шага — убедиться, что некоторые (хоть какие-нибудь) данные могут пройти через всю программу. Мы могли бы реализовать (кодированием по нисходящей схеме) некоторую часть программы РЕДАКТИРОВАТЬ, достаточную, например, для того, чтобы она могла принимать сообщения только типа А и притом, если они предварительно упорядочены и не содержат ошибок. То есть мы можем допустить, что ни одна из операций по редактированию и сортировке в модуле РЕДАКТИРОВАТЬ не реализована и что он даже не может считывать сообщения типов B, C и D. Аналогично программа модуля ОБНОВИТЬ должна быть спроектирована и написана вплоть до такого уровня, который необходим для распознавания сообщений типа А и чтения «старых» записей главного файла. Мы могли бы допустить, что программа ОБНОВИТЬ не может даже распознавать сообщения других типов или правильно обрабатывать сообщения типа А. Все, что она пока может делать, это — считывать сообщения типа А и далее игнорировать их, а также считывать записи входного главного файла и тут же пересылать их в выходной главный файл. Точно так же мы могли бы написать программу ВЫБРАТЬ таким образом, чтобы она выполняла считывание записей главного файла и, возможно, извлекала для последующего упорядочения записи типа X (поскольку большинство вычислительных систем располагает пакетом стандартных подпрограмм упорядочения, мы могли бы на этом этапе включить в реализацию модуля также и операции сортировки). Выходные данные программы ВЫБРАТЬ могли бы считываться некоторой фиктивной реализацией программы печати отчетов, которая бы просто выдавала эти записи на печатающее устройство.

3. Следующим шагом полезно было бы расширить все программы настолько, чтобы модуль РЕДАКТИРОВАТЬ мог воспринимать сообщения всех типов, при условии, что в них нет ошибок и что они предварительно упорядочены. Аналогично мы.'можем ожидать, что модуль ОБНОВИТЬ уже различает сообщения всех типов и выполняет вызовы соответствующих обрабатывающих модулей, причем

реализации этих модулей могут пока оставаться фиктивными. Мы могли бы ожидать, что программа ВЫБРАТЬ должна быть относительно простой в сравнении с другими программами, поэтому можно было бы воспользоваться фиктивными реализациями *всех* программ, формирующих отчеты, поскольку острой необходимости в фактической выдаче отчетов на печать может не быть (предполагается, что предварительная версия формы отчета не должна быть предъявлена для утверждения пользователю программы).

4. Мы настолько продвинулись, что на следующем шаге может быть следовало бы расширить программу РЕДАКТИРОВАТЬ, наделив ее способностью сортировать сообщения, которые она может принимать. Мы могли бы по-прежнему не беспокоиться о редактировании сообщений, а это означает, что для испытаний программист должен использовать только «чистые»сообщения. Аналогично программу ОБНОВИТЬ следовало бы расширить настолько, чтобы она могла контролировать последовательность сообщений.

5. Следующим логическим шагом нужно было бы дописать необходимые коды программы РЕДАКТИРОВАТЬ, выполняющие все операции по редактированию сообщений типа A; сообщения типов B, C и D можно было бы по-прежнему пропускать без редактирования. Аналогично мы могли бы затем дописать коды программы ОБНОВИТЬ, выполняющие реальную обработку A-сообщений. На этом шаге можно было бы не делать каких-либо изменений в модуле ВЫБРАТЬ и все еще не писать окончательной версии программных модулей печати отчетов.

6. Добившись правильной обработки A-сообщений, мы могли бы затем дописать коды программ РЕДАКТИРОВАТЬ и ОБНОВИТЬ, выполняющие те же операции над B-, C- и D-сообщениями.

7. Наконец, мы могли бы написать коды программы, печатающей отчеты. Быть может, это было бы удобнее сделать несколько раньше, если бы оказалось, что простая печать неотредактированных выходных файлов модуля ВЫБРАТЬ (о чем говорилось выше при описании шага 2) трудна в интерпретации.

Одним из удобных свойств нисходящего тестирования является то, что обычно его можно планировать заранее, т. е. для большинства типов программ не очень трудно определить стадии процесса тестирования, как это было сделано в рассмотренном выше примере. Главное что здесь следует понять, состоит в том, что это планирование может быть осуществлено в самом начале разработки. Действительно представляется нетрудным делом составить график этапов тестирования, т. е. в самом начале разработки мы могли бы учесть дополнительные объемы работ, которые необходимо выполнить на каждом этапе, и определить, что, скажем, первый этап завершится к 1 января, второй — к 1 февраля и т. д. В нескольких программных разработках такое планирование оказалось вполне успешным.

2.3.2. Достоинства нисходящего тестирования

Некоторые программисты находят методологию нисходящего тестирования интуитивно естественной; некоторые отмечают, что фактически они давно уже пользуются этой методологией. В то же время в некоторых организациях саму эту концепцию считают довольно радикальной и не могут понять, почему кому-то может прийти в голову применить нисходящее тестирование на практике. В нескольких программных разработках были обнаружены следующие преимущества нисходящего тестирования.

1. Тестирование на уровне системы в классическом его понимании фактически упраздняется.
2. Основные интерфейсы испытываются в первую очередь. В результате наиболее серьезные ошибки выявляются на ранних этапах разработки, в то время как незначительные упущения обнаруживаются на более поздних этапах.
3. Пользователей можно ознакомить с предварительными версиями программы на относительно ранней стадии разработки.
4. Если оказывается невозможным завершение разработки всей программы к назначенному крайнему сроку, то вероятно завершение некоторой, допускающей независимое использование, части программы.
5. Использование методологии нисходящего тестирования часто значительно облегчает выявление ошибок программирования (т. е. в отладке в отличие от тестирования).
6. Затраты времени на тестирование на протяжении разработки распределяются более равномерно, что исключает повышенную потребность в машинном времени на заключительных этапах разработки.
7. Для программистов психологическая атмосфера значительно менее напряжена, поскольку они могут видеть результаты успешного тестирования логического скелета окончательной программы.
8. В нисходящем тестировании обеспечивается реальная «тестовая обстановка» испытаний модулей более низких уровней.

Исключение тестирования на уровне системы. Как было замечено в нескольких программных разработках, тестирование на уровне всей системы может занимать по времени от одной трети до половины всей разработки. Важно отметить, что на протяжении этого промежутка не наблюдается сколько-нибудь существенного и ощутимого продвижения в работе. Программисты доказывают, что разработанные ими модули работают правильно (или по крайней мере *утверждают,* что они выполнили соответствующие испытания) и тем не менее вся система оказывается неработоспособной. Человеку, не имеющему отношения к ЭВМ (например, пользователю, а возможно, и руководителю разработки), не очевидно, что программисты действительно что-то *делают* в этот период. И если этим непосвя-

щенным может казаться, что программисты *никогда* не делают чего-либо реального, такое отношение к программистам в этот «смутный» период испытаний системы может приобрести значительный вес. Мы не должны забывать, что испытание системы в некоторых разработках длилось от двух до трех *лет* и что в течение этого периода отношение к разработке в целом может быть подвержено различным конъюнктурным влияниям, не исключающим отказ от всего проекта.

До некоторой степени эта трудность может быть разрешена использованием нисходящего тестирования. Ключевые события, легко интерпретируемые результаты которых способны осознать самые неискушенные в области программирования, могут быть довольно точно спланированы во времени. И если даже в нисходящем тестировании требуется столько же времени, сколько расходуется в классическом подходе (обычно это не так), что мы имеем к тому же график важнейших ключевых событий, охватывающий все этапы тестирования системы.

Важнейшие интерфейсы тестируются в первую очередь. Утверждение, что если бы в наших программах не было ошибок, то мы не нуждались бы в их тестировании, является, очевидно, труизмом На самом деле мы *делаем* ошибки, одни из них существенные, другие — относительно пустячные. Разумно считать, что если мы допустили серьезную логическую оплошность, которая может повлиять на структуру проекта всей нашей программы, то нам следует выявить ее как можно раньше. Если мы допустили тривиальную ошибку при написании кодов программы, которая может повлиять только на один модуль, то довольно безразлично, когда именно мы выявим эту ошибку. Методология нисходящего тестирования способствует именно такой логике организации испытаний, тогда как восходящее тестирование способствует утверждению как раз противоположных представлений.

Рассмотрим в качестве примера гипотетическую программу, показанную на рис. 2.11. В классической восходящей схеме разработки одному программисту, возможно, было бы предложено разработать модуль EDIT, другому — модуль UPDATE. Определив интерфейсы между этими модулями, теоретически каждый программист мог бы продолжать работу независимо. К сожалению, может случиться, что спустя несколько месяцев работы они обнаружат, что допустили ошибку в понимании смысла переменных интерфейса, и как следствие вполне вероятно, что модуль UPDATE окажется не в состоянии выполнить чтения выходных файлов модуля EDIT. Хотя такая ситуация и может показаться исключительной, она *наблюдалась* в нескольких разработках и нередко после года или большего срока работы десятков программистов! В одной такой крупной разработке, предпринятой одной страховой компанией в Австралии, от проекта пришлось отказаться после двух лет работы

по той причине, что две основные его подсистемы (спроектированные и написанные порознь по восходящей схеме) оказались совершенно неспособными к взаимодействию друг с другом[1)].

В случае применения методологии нисходящего тестирования попытки опробовать эти важнейшие интерфейсы предпринимаются как можно раньше. Напомним, что первое действие в тестировании примера по рис. 2.11 сводилось к проверке правильности составления операторов управления заданиями. Как мы в свое время отмечали, ошибки этого этапа оказываются причиной многих и серьезных проблем. На следующем этапе тестирования предъявляются минимальные требования к взаимодействию между программами EDIT, UPDATE, SELECT и программами печати отчетов; здесь нужно убедиться только в том, что если программа EDIT пишет на выходе двухсотсимвольные записи, то программа UPDATE действительно может читать двухсотсимвольные записи на входе, не проверяя правильности данных, составляющих эти записи. Вспомним также, что на четвертом шаге тестирования требовалось, чтобы программой EDIT выполнялась сортировка проходящих через нее сообщений. И поскольку, таким образом, программе UPDATE через интерфейс должны передаваться упорядоченные сообщения, есть смысл на входе в программу UPDATE выполнить *проверку* того, что эти сообщения действительно упорядочены. Если на относительно поздней стадии тестирования (например, на шаге 5 или 6) в модуле UPDATE была обнаружена ошибка, у нас была бы относительная уверенность в том, что ееможно исключить, не внося никаких изменений в программу EDIT.

Пользователей можно ознакомить с предварительной версией всей программы. Многим программистам сразу ясно, что в процессе нисходящего тестирования программы весьма желательна возможность представить пользователям предварительную версию программы. Если разработка системы отстает от графика или же пользователь сомневается в том, что она может быть завершена в срок, доступная восприятию широкой аудитории предварительная версия программы может оказаться весьма важным средством урегулирования отношений. Аналогично, если пользователь не вполне представляет, что именно ему нужно, то было бы крайне полезно как можно раньше ознакомить его с предварительной версией всей программы. Высказанные им оценки, возможно, привели бы к тому, что многие модули нижнего уровня проектировались бы иначе, чем в других условиях.

Верно, что обычно оказывается необходимым представить поль-

[1)] Дж. Миили, принимавший непосредственное участие в разработке первых версий математического обеспечения ОС Системы IBM/360, сделал очень проницательное замечание по этому поводу, указав, что характер интерфейсов между основными подсистемами системы обычно отражает характер взаимодействия между организациями, разрабатывающими эти подсистемы.

зователям некоторые *результаты,* дабы убедить их в том, что программа делает именно то, что от нее хотят. Так, в примере, приведенном на рис. 2.11, было бы необходимо иметь детально разработанными программы печати выходных отчетов. Это позволило бы убедить пользователя в том, что программы выполняют значительный объем работы. Многие программисты, правда, замечают, что пользователи склонны к излишней разборчивости в отношении форматов выходных данных. Однако выполнив эту часть работы, мы действительно смогли бы показать, что определенные типы сообщений обрабатываются правильно, и это — задолго до того, как будут спроектированы и запрограммированы различные операции по редактированию. Из примера на рис. 2.11 должно стать ясным, что, когда мы планируем в самом начале разработки этапы тестирования, программист ставится в такие условия, что будет подвергаться большему влиянию со стороны пользователя.

Проблемы завершающего этапа иногда могут быть упрощены. Читая курсы лекций и проводя семинары для многих тысяч программистов из разных стран, я не переставал удивляться тому количеству разработок, которые выполнялись по графикам, оцениваемым программистами как совершенно нереалистичные. «Нам дали на выполнение этой работы шесть месяцев,— говорит программист,— в то время как совершенно ясно, что на нее потребуется не меньше трех лет». Участвуя в качестве консультанта во многих из таких разработок, я склонен принять точку зрения программистов; *действительно* бывает очевидным, что на разработку потребуется значительно больше времени, чем то, которое отпущено руководством. Несмотря на все разумные суждения по поводу этой нелепой ситуации, мы должны констатировать, что она *действительно* имеет место. В большом числе случаев разработка организуется таким образом, что программист оценивает свое отставание от графика в полгода уже в первый день работы над проектом! Даже в том случае, когда график выглядит обоснованным, всегда имеется некоторая вероятность того, что реальный ход разработки в него не уложится.

Имея в виду эти потенциальные проблемы, использование принципа «сверху вниз» может способствовать тому, что крупная политическая катастрофа обернется небольшим замешательством. Если пользователь считает, что проект может быть закончен за шесть месяцев, а программист уверен, что на него потребуется год, то важно знать ответ на вопрос: какая часть проекта будет выполнена за шесть месяцев? Пользователь, некомпетентный в вопросах применения ЭВМ, бывает озадачен, услышав следующий ответ: «Я закончил составление кода программы и тестирование некоторых модулей»; в конце концов он может не знать, что такое модуль, не говоря уже о тестировании модулей. В то же время, если мы скажем: «Программа может обрабатывать А-сообщения при условии,

что вы не будете посылать в нее ошибочные входные данные; мы закончим работы по B-, C- и D-сообщениям в следующем месяце»,— то он, возможно, поймет, о чем идет речь. Ясно, что он будет попрежнему неудовлетворен тем, что мы не закончили в срок всю программу, но по крайней мере у него есть хоть *что-то,* чем он может пользоваться (конечно, требуется весьма тщательная оценка выбора тех важнейших функций, которые должны быть реализованы в первую очередь).

Нисходящее тестирование упрощает отладку. Хотя это может показаться неочевидным, большинство программистов считают, что методология «сверху вниз» значительно облегчает отладку. Очень важно, что в данном случае мы делаем различие между *отладкой* и *тестированием* (подробному обсуждению этого посвящены гл. 7 и 8). В этом контексте мы полагаем, что программист осознает наличие ошибки в его программе и что основная задача состоит в локализации и определении характера этой ошибки.

Чтобы уяснить трудности отладки, рассмотрим классическую восходящую схему отладки примера, ранее иллюстрированного рис. 2.9, *а—в*. Если в тестировании модулей программы рис.2.9,а обнаруживается какая-то ошибка, то программист обычно не сталкивается с особыми трудностями в ее идентификации. В конце концов, так как речь идет о тестировании модулей, мы можем предположить, что всякая проявившаяся ошибка находится в программе того модуля, который в этот момент тестируется (при этом, очевидно, не учитываются возможные ошибки в самих тестовых данных и разные другие детали, которые иногда превращают отладку в настоящий кошмар). Когда, однако, мы приступили к тестированию на уровне подсистем, иллюстрируемому рис. 2.9, б, ситуация оказывается значительно сложнее. Здесь впервые объединяются в целое три или четыре модуля. Если имеется какая-нибудь ошибка в любом из модулей, то ее характер может быть неочевиден программисту, особенно если модули были написаны разными программистами! Тот факт, что причина ошибки может таиться в любом из модулей, делает отладку намного труднее — мы просто не знаем, с чего начать ее поиск. Ситуация еще более усложняется на стадии тестирования системы, иллюстрируемой рис. 2.9, *в*. Здесь мы можем иметь дело с сотнями или даже тысячами взаимодействующих друг с другом модулей, и каждый из них способен оказаться источником ошибки, характер которой может быть неочевидным для программиста (программистов). И в этом случае причиной трудностей является то, что большое число модулей оказалось впервые объединенным в систему, и если один из них не работает, то может быть не очевидно, *какой* именно.

Подход «сверху вниз» приводит к значительно меньшим трудностям. В большинстве случаев в начале мы имеем некоторый логический скелет программы, о котором известно, что он правильно

исполняет определенные функции; тестирование сводится к тому, что мы добавляем к системе один новый модуль и выясняем, правильно ли она работает. Если по некоторой причине новая версия нашей программы не работает, то разумно предположить (хотя иногда это и не так), что источником ошибки является либо новый модуль, либо его интерфейс с уже существующим модулем более высокого уровня, либо сам этот модуль более высокого уровня. Мы пользуемся преимуществами управляемого научного эксперимента: мы начинаем с некоторого изученного числа элементов и добавляем к ним один новый (именно новый модуль низкого уровня, который подлежит тестированию). Если эта новая версия программы не работает, то довольно ясно, где искать соответствующую ошибку. В худшем случае мы всегда можем удалить неработающий модуль и восстановить прежний логический скелет программы!

Лучшее распределение времени тестирования. Некоторые программисты и руководители проектов выражают озабоченность в связи с тем, что применение нисходящего тестирования может привести к возрастанию необходимого для тестирования времени. Причина такой озабоченности вполне понятна; на первый взгляд представляется, что каждый раз, когда мы добавляем к существующему логическому скелету программы новый модуль, мы тестируем целиком всю систему. Эти сомнения, однако, не имеют под собой никаких практических оснований; в большинстве случаев методология нисходящего тестирования требует для тестирования времени существенно меньше, и, во всяком случае, время испытаний на протяжении разработки распределяется более равномерно.

Нисходящее тестирование способствует оздоровлению психологической атмосферы. В крупных программных разработках не следует игнорировать психологическое состояние программистов. Чрезвычайно трудно в течение месяцев (или лет) заниматься проектированием и написанием программы, не имея возможности оценить значимость достигнутых результатов, особенно если разработка находится под угрозой отмены по той или иной причине, так как пользователь и (или) руководство считают, что она слишком отстает от графика! Прогресс в разработке системы, создаваемой по принципу «снизу вверх», часто бывает трудно измерить даже на стадии тестирования; это приводит к ухудшению психологической атмосферы, в которой ведется разработка.

В то же время в случае разработки, ведущейся по принципу «сверху вниз», определяются наиболее значительные цели, которые достигаются на ранних этапах разработки. Все программисты (не говоря уже о пользователях и руководстве) могут удостовериться в том, что их программа действительно выполняет именно те процедуры, которые были обещаны. Улучшение психологической атмосферы часто способствует более энергичной работе по достижению очередного промежуточного результата и т. д.

В нисходящей схеме обеспечиваются естественные условия испытаний. В большинстве случаев чрезвычайно трудно тестировать модуль изолированно. Для всякого модуля необходимы некоторые входные данные, посылаемые откуда-то извне; часто ему требуются записи базы данных, может оказаться необходимым доступ к различным таблицам, управляющим блокам и другим внешним данным. Исходя из схемы «снизу вверх», программист часто вынужден *имитировать* то окружение, в котором, как ожидается, должен функционировать его модуль; иногда для этого специально формируют тестовую среду модульных испытаний (более подробно эти вопросы обсуждаются в разд. 7.8.3).

С использованием нисходящего тестирования существующий логический скелет программы обеспечивает естественные условия испытаний, в которых может функционировать вновь составленный модуль. Конечно, модули уровней, расположенных *ниже* тестируемого, все еще представлены лишь «заготовками» программ; тем не менее мы могли бы рассчитывать на то, что к моменту начала функционирования нашего нового модуля для него была бы обеспечена его внешняя среда, и это значительно облегчает процесс тестирования. До некоторой степени здесь необходимо руководствоваться здравым смыслом. Может оказаться разумным *прежде,* чем вносить новый модуль в существующий логический скелет программы, убедиться в том, что он, по крайней мере выполняясь от начала до конца, не выбрасывается операционной системой.

В некоторых случаях программисты настаивают на том, что тестировать модули по схеме «сверху вниз» *много* труднее. Их доводы основаны на том, что в этой схеме для каждого вновь вводимого модуля соответствующие тестовые данные должны формироваться на входе всей программы, например в нашем случае, отвечающем рис. 2.11, нужно было бы формировать соответствующие А-сообщения. В некоторых случаях может оказаться очень трудной задачей формирование всех таких вариантов А-сообщений, которые в конце концов породили бы всевозможные варианты обращений к нашему новому модулю нижнего уровня. Некоторые программисты считают, что было бы намного легче использовать генератор тестовых данных, который может формировать всевозможные входные данные для нашего нового модуля. Нет ничего предосудительного в таком подходе; однако, после того как закончено тестирование модуля с использованием генератора тестовых данных, новый модуль все же следует внести в логический скелет программы, чтобы убедиться, что там формируются необходимые входные данные, которые новый модуль готов принять.

2.3.3. Модификации нисходящего тестирования

Как мы отметили в предыдущем разделе, программисты иногда затрудняются в применении нисходящей схемы тестирования в «чистом виде». Мы должны еще раз напомнить, что излагаемая здесь методология не должна рассматриваться как некий свод жестких правил, ее следует рассматривать как *основной принцип,* допускающий необходимые небольшие вариации в соответствии с конкретными особенностями проектов.

Например, мы указывали, что в своей крайней форме нисходящая схема тестирования предполагает, что после того, как проектирование главной программы завершено, следует приступить тут же к ее кодированию и тестированию, прежде чем будет создан *хотя бы один* из модулей более низких уровней. Не похоже, чтобы это означало сколько-нибудь существенное продвижение, поскольку трудно добиться выполнения чего-либо от программы, *все* модули которой фиктивны (хотя, с другой стороны, это позволяет нам убедиться в том, что наши карты управления заданиями составлены правильно — работа, заслуживающая во многих проектах серьезного внимания). В большинстве случаев мы требуем, чтобы были реализованы главная программа и один или два более низких уровня модулей, прежде чем станет возможным сколько-нибудь нетривиальное тестирование.

На самом деле может возникнуть такая ситуация, когда требуется почти окончательная реализация некоторых частей программы, в то время как другие важнейшие компоненты по-прежнему представлены фиктивными модулями. Рис. 2.12 иллюстрирует пример такого рода. Модуль А и все его подчиненные модули спроектиро-

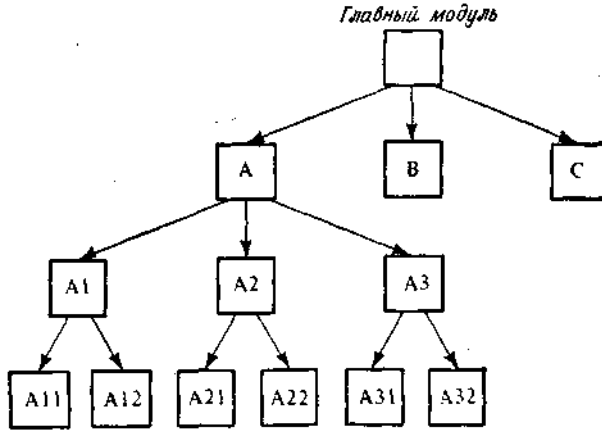


Рис. 2.12. Вариант нисходящего тестирования.

ваны, закодированы и испытаны, в то время как модули B, C и D остаются в виде заготовок. Простым примером такого подхода может служить разработка модулей ввода-вывода с неопределенными подпрограммами обработки, оставляемая часть соответствующей программы не может вообще выполнять каких-либо действий! Аналогично в примере рис. 2.11 оказалось, что модуль SELECT следует разработать довольно подробно, в то время как модули EDIT, UPDATE и модули печати отчетов все еще остаются довольно схематичными.'

Те же самые методологические вопросы допускают несколько иное толкование. Иллюстрируя концепцию нисходящего тестирования в начале разд. 2.3.1, мы использовали рис. 2.10, *а—в,* дабы подчеркнуть, что модули разрабатываются и испытываются уровнями. Некоторые программисты понимают это так, что они должны строго следовать правилу «уровень за уровнем», т. е., например, *все* модули уровня 2 должны быть спроектированы, закодированы и испытаны, прежде чем будет спроектирован хоть один модуль уровня 3.

В принципе мы действительно предполагаем именно это; на практике, однако, иногда бывает необходимо или желательно прибегнуть к «челночным» проходам в структуре программы. Из рис. 2.13, а, например, можно видеть, что программист реализовал главную программу и написал коды модулей A, B, C и D; все модули более низкого уровня фиктивные. Из рис. 2.13, *б* мы, однако, видим, что он дописал два дополнительных уровня к модулю A, в то время как модули B, C и D остались изолированными. Мы полагаем, что такой порядок продиктован практическими соображениями. Далее, как показывает рис. 2.13, *в,* программист расширил на один уровень модули B, C и D и затем, как это иллюстрируется рис. 2.13, счел целесообразным дописать еще два уровня подмодулей модуля D. Основной подход при этом не изменяет нисходящей схеме тестирования, подобные «челночные» вариации при аккуратном применении не должны приводить к каким-либо затруднениям.

Мы можем также столкнуться с необходимостью частичного тестирования по восходящей схеме. Такая возможность объясняется несколькими причинами.

1. Недостаток необходимого количества времени для тестирования может вынудить программиста прибегнуть к независимому испытанию модуля «на скорую руку». Убедившись, что модуль по крайней мере исполняется от начала до конца, он включает его в логический скелет программы для нисходящего тестирования.

2. Как мы уже отметили в предыдущем разделе, программист может найти затруднительным формирование необходимого набора входных данных для его модуля, действуя в соответствии с принципом «сверху вниз». Может оказаться намного проще использовать генератор тестовых данных и провести независимое испытание модуля.

3. Недостаток необходимого времени для тестирования может помешать программисту провести в полной мере тестирование по нисходящей схеме. В примере рис. 2.11 программист мог бы добавить новый модуль низкого уровня к программе UPDATE. В идеальном случае, чтобы убедиться в правильности этого нового модуля, ему следовало провести испытания, посылая сообщения в программу UPDATE через программу EDIT. Однако, если он не располагает достаточным временем на тестирование, он может решить использовать для испытания программы нового модуля UPDATE один из упорядоченных файлов (т. е. выходных файлов программы EDIT) одного из предыдущих тестовых прогонов.

4. Проблемы организации управления и укомплектования кадров (которые мы более подробно обсудим в разд. 2.4) могут привести к тому, что некоторое количество младших программистов окажется занятым разработкой модулей нижних уровней прежде, чем будет завершено написание логического скелета верхнего уровня программы. Таким образом, появляется необходимость в некоторых независимых испытаниях модулей и объединении их по схеме «снизу вверх», прежде чем будет готов логический скелет верхнего

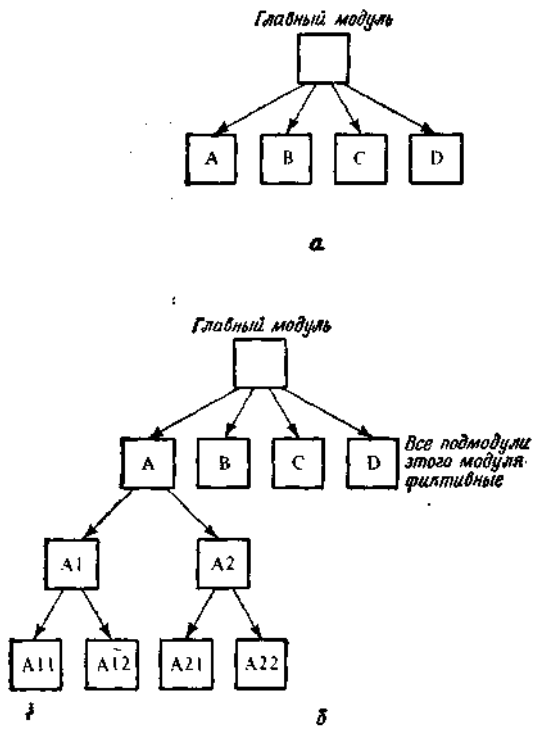
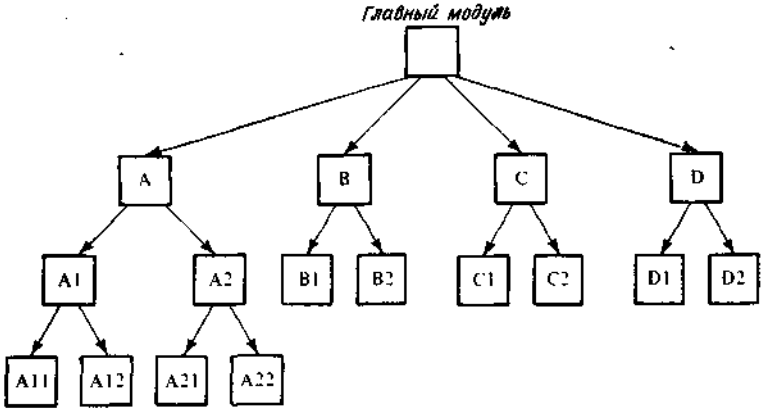
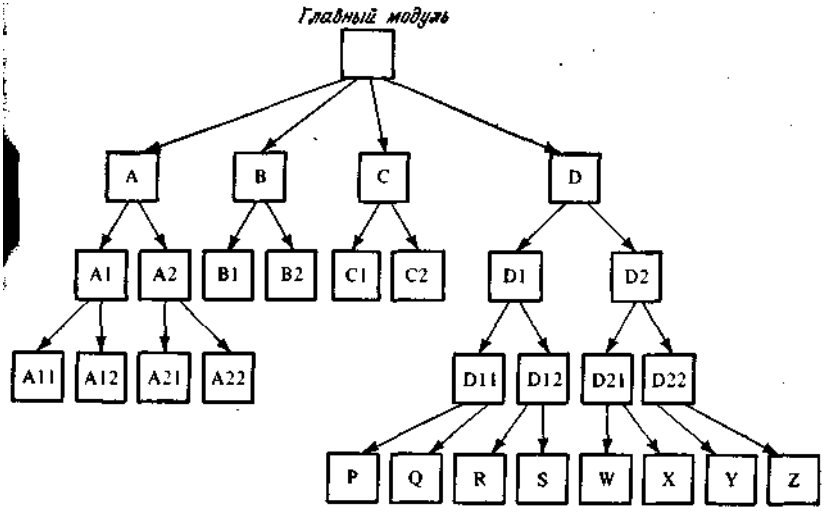


Рис. 2.13. «Челночный» проход при разработке по нисходящей схеме.





а

Рис. 2.13. (Продолжение.)

уровня программы; к сожалению, ситуации такого рода часто приводят к тестированию полностью по восходящей схеме. Повторяем, что приведенные причины носят *практический* характер. У программиста может быть намерение провести тестирование по нисходящей схеме, но он не может как-либо влиять на количество машинного времени, представляемого ему для тестирования.

2.3.4. Структурированный разбор

Одним из наиболее интересных вариантов нисходящего тестирования является метод *структурированного разбора,* возникший как часть разработанной на фирме IBM методологии разработки проектов, связываемой с выражением «бригада главного программиста». В других организациях этот метод известен под названием «бригадная отладка» и многими другими. Структурированный разбор можно представить себе как ряд формализованных процедур по анализу — целой бригадой программистов — требований к программе, ее проекта, кодовой реализации и адекватности испытаний. При должной организации структурированный разбор осуществляется параллельно и в согласии с теми выполняемыми по нисходящей схеме процессами, которые рассмотреныв этой главе, а также процессами структурного программирования, составляющими предмет рассмотрения гл. 4.

Как мы уже установили, проектирование и написание программ следует выполнять по нисходящей схеме. Это означает, что программист должен завершить проектирование и кодирование верхнего уровня логической схемы программы на сравнительноранних стадиях разработки. По завершении проектирования и кодирования очередного уровня программы программист может представить выбранные решения для коллективногорассмотрения группой программистов, занятых в той же разработке, и (или) других программистов той же организации. Такой коллективный анализ позволяет «пройтись» по всей программе и последовательно, шаг за шагом, убедиться в том, что

1. Логическая схема действительно верна, т. е. что она соответствует назначению программы.

2. Логическая схема согласуется с соответствующими стандартами данной организации, т. е. она достаточно проста, целесообразна, имеет модульную организацию, структурирована (в смысле представлений, излагаемых в гл. 4), хорошо документирована и т. д.

С момента появления ЭВМ программистам приходилось проделыватьтщательную «настольную» проверку программы, прежде чем приступить к прогнозам ее на машине; со временем, однако, необходимость в «настольной» проверке становилась все меньшей, по мере того как в 60-х гг. машинное время становилось все более доступным. В некоторых организациях с введением режима разделения времени само понятие «настольной» проверки вовсе вышло из употребления. Структурированный разбор является в некотором смысле возвратом к прежней методологии «настольной» проверки с тем, однако, существенным отличием, что теперь некоторая *группа* программистов должна рассматривать проект и коды программы одного программиста.

Истинная ценность метода структурированного разбора заключается в том, что он связывается с проектированием и кодированием по нисходящей схеме. На самых ранних этапах программной разработки программист должен вынести на рассмотрение своих коллег или других членов бригады разработанный им проект *всей* его программы. В значительной мере программа может быть представлена в форме фиктивных модулей, но общая логическая схема и модульная структура всей программы должны иметь законченный вид. Цель коллективного рассмотрения состоит в том, чтобы убедиться в правильности общей логической схемы программы, *предполагая, что фиктивные модули, когда их фактически реализуют, будут работать правильно.* В начале разработки бригада интересуется не столько подробностями реализации модулей нижних уровней, сколько сосредоточивает свое внимание на анализе верхнего уровня структуры программы, выявляя таким образом наиболее серьезные упущения, которые может не осознавать программист-разработчик.

В нескольких организациях методология структурированного разбора формализована и внесена составной частью в общие руководства по управлению разработками. Руководство и управление разработками не относится к основным вопросам этого учебника; не задерживаясь на них более, мы лучше перечислим некоторые из наиболее интересных процедурных аспектов метода структурированного разбора.

1. Как один из членов бригады программистов, занятых в одном и том же проекте, программист должен предусмотреть структурированный разбор его программы, назначив заранее официальное заседание. За день или за два до заседания программистам группы передаются копии проекта и (или) текста программы, и (или) тестовых данных, с которыми они должны ознакомиться к началу заседания.

2. Присутствие на структурированных разборах руководителя проекта, как правило, не допускается. Во многих организациях считают, что присутствие руководителя проекта (на ответственности которого скорее административная, а не техническая сторона дела) сдерживало бы откровенный и свободный обмен мнениями о рассматриваемой программе.

3. В некоторых случаях необходимо предусмотреть участие в обсуждении стороннего наблюдателя, который мог бы разрешить возможные споры программистов. Иногда возникает опасность, что слишком агрессивно настроенная группа программистов может«разнести» до основания предлагаемый проект программиста-разработчика, который никогда не может быть абсолютно уверен в правильности своей работы; в других случаях слишком агрессивный программист может подавить критику и предложения остальной части бригады. В действительности некоторые группы пришли к выводу,

что наряду с решением технических вопросов не менее важно уделять внимание вопросам межличностных отношений, вынося их на коллективное обсуждение.

4. В ходе заседания ведется официальный протокол (часто это делает программист, ведающий библиотекой), копии которого все могут получить на следующий день. Программист, работа которого явилась предметом структурированного разбора, обязан представить официальный ответ на критику и предложения группы.

5. В структурированных разборах всячески *избегают* проектирования программы заново. Цель заседания в том, чтобы выявить трудности и (или) убедиться в правильности работы программы. Если в программе обнаружена ошибка, то предполагается, что программист должен ее исправить. Попытки группового экспромта в проектировании обычно кончаются явной неудачей.

6. Вся бригада несет ответственность за ошибки или упущения, которые могут обнаружиться в программе после того, как она передана пользователю. До известной степени вина ложится на программиста-разработчика, который впервыедопустил появлениеошибки в программе, но вина ложится и на всю группу, просмотревшую такую ошибку в структурированном разборе. Таким образом, поощряется внедрение идеологии «неиндивидуального программирования» Вейнберга, выдвинутой им в книге The Psychology of Computer Programming, D. Van Nostrand Co., 1971.

При первом знакомстве с методом структурированного разбора многие программисты выражают к нему довольно негативное отношение. Их прежде всего беспокоит то, что этот метод приводит к замедлению разработки: им кажется слишком расточительным занимать пять или шесть программистов разбором работы одного из них. На самом деле все обстоит как раз наоборот. Метод структурированного разбора дает значительную экономию времени. Прежде всего нужно заметить, что требуется всего час или два на рассмотрение недельной работы программиста; по крайней мере столько времени ему понадобится на проектирование и кодирование (после того как он выбросит несколько предварительных вариантов). Более важно то, что, представляя проект и коды программы на рассмотрение пяти или шести человек, каждый из которых смотрит на вещи с несколько иной точки зрения, программист значительно повышает вероятность того, что большая часть ошибок будет исключена еще до ввода программы в машину, включая и те скрытые ошибки, на выявление которых он потратил бы не один день работы такими методами отладки, как трассировка, анализ мгновенного состояния программы и т. п.

Представляется, что *действительной* причиной отрицательного отношения к методу структурированного разбора является то, что они предпочитают не показывать свою программу другим. «Чтение чужих программ,— сказал мне один английский программист,—

похоже на чтение личных писем — в приличном обществе это не принято». В самом деле, многие программисты рассматривают структурированные разборы как практику, травмирующую самолюбие, а некоторые ее просто не приемлют. В огромной степени успех или неудача применения этого метода может зависеть от личных качеств программистов, работающих в одной бригаде. Если межличностные конфликты удается исключить или минимизировать и программистов можно заставить отказаться от излишнего самолюбия в отношении «их» программ, применение метода структурированного разбора в сочетании с принципами нисходящего проектирования обычно оказывается весьма успешным. Например, одной бригаде в Австралии удалось в течение трех месяцев не допустить ни единой ошибки, т. е. к тому моменту, когда программа была фактически введена в машину, в ней уже не осталось ни одной ошибки.

2.4. Альтернативы, варианты и трудности нисходящего проектирования

В предыдущих разделах мы подробно рассмотрели принципиальные положения нисходящей схемы проектирования, кодирования и тестирования. Важно иметь в виду, что мы обсуждали именно *принципы,* а не какой-нибудь свод правил проектирования программ. Как мы уже неоднократно отмечали, многие программисты могут время от времени обнаруживать необходимость несколько отклониться от идеальной нисходящей схемы проектирования, кодирования и тестирования. Настоящий раздел имеет целью обсудить характер такого рода компромиссов и причин, которыми они вызваны. Мы хотели бы изучить прежде всего следующие темы.

1. Традиционный подход к разработке, сочетающий нисходящег проектирование с неупорядоченным кодированием (т. е. различными комбинациями схем «сверху вниз» и «снизу вверх») и восходящим тестированием.
2. Оправданностью в ряде случаев сочетания нисходящего и восходящего принципов проектирования.
3. Предпочтение, оказываемое некоторыми программистами проектированию программы по нисходящей схеме, с выполнением кодирования или тестирования также по нисходящей схеме, лишь после полного завершения проектирования до самого последнего уровня.
4. Предпочтение, оказываемое некоторыми программистами выполнению параллельного нисходящего проектирования и кодирования с последующим переходом к тестированию, также осуществляемому по нисходящей схеме после завершения кодирования.
5. Трудности управления и организации, способные помешать руководителю проекта рационально распределить кадры для реа-

лизации нисходящей схемы проектирования, кодирования и тестирования.

6. Задержки в поставке ЭВМ и недостаток времени для тестирования, способные воспрепятствовать применению принципов нисходящего проектирования, кодирования и тестирования во всей их полноте.

2.4.1. Традиционный подход

Как мы уже упомянули в этой главе, традиционный подход к разработке программы представляет собой некоторое сочетание нисходящего проектирования, неупорядоченного кодирования и восходящего тестирования. Многие программисты сообщают, что, хотя они лично убеждены в разумности методологии нисходящего тестирования, они не могут убедить пользоваться ею других программистов организации. Тот факт, что тестирование всегда проводилось по восходящей схеме, в некоторых организациях оказывается достаточно сильным основанием того, чтобы ее не менять.

После обсуждения на протяжении всей этой главы основных идей нисходящей схемы у нас нет необходимости повторять доводы против схемы «снизу вверх». Здесь важно подчеркнуть то, что восходящая схема используется более широко и что инерция некоторых крупных компаний служит препятствием для каких-либо изменений.

2.4.2. Сочетания нисходящей и восходящей схем проектирования

Во многих случаях представляется разумным сочетать восходящую и нисходящую схемы проектирования. Вот два наиболее распространенных примера.

3. Программист, зная о существовании некоторого числа полезных прикладных подпрограмм, стремится (сознательно или бессознательно) так проектировать свою программу, чтобы употребить их.

2. На относительно ранней стадии проектирования своей программы программист предвидит, что некоторые простые или стандартные модули потребуются нескольким различным частям программы. Примерами таких модулей могут служить подпрограммы анализа ошибок, ввода-вывода, редактирующие подпрограммы и подпрограммы просмотра таблиц. В обоих случаях программист мог бы *начать* проектирование по восходящей схеме; вскоре, однако, он оказывается погруженным в проектирование нескольких модулей нижнего уровня, как это иллюстрируется рис. 2.14, *а*. Далее он продолжает проектирование по обеим схемам «снизу вверх» *и* «сверху вниз», как показано на рис. 2.14, *б*. В альтернативном варианте, реализовав несколько модулей нижнего уровня, програм-

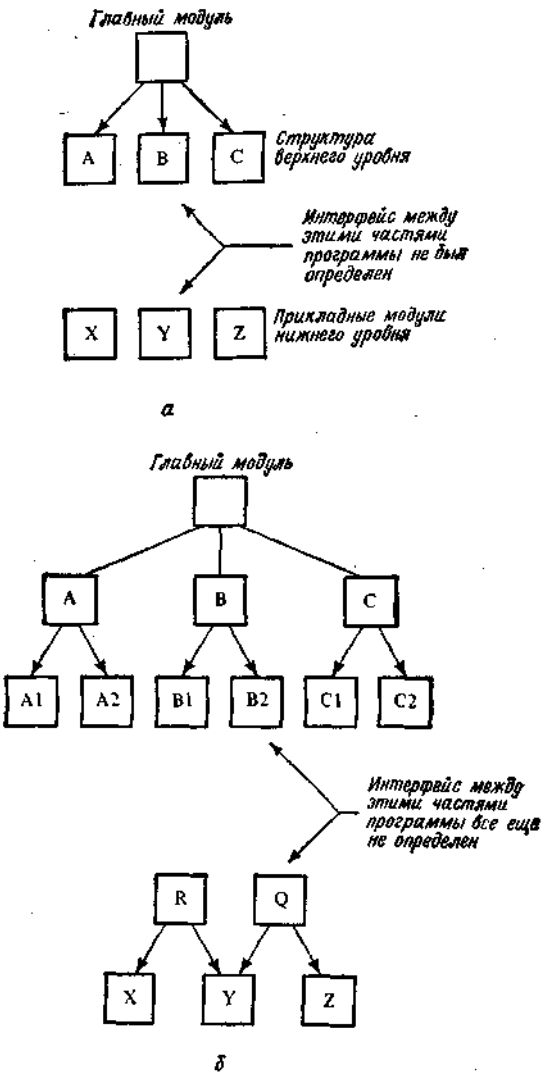


Рис. 2.14. Нисходящее и восходящее проектирование.

мист достраивает остальную часть программы в *направлении нижних* . модулей, обеспечивая их совместимость.

Нет никаких оснований считать порочным такое использование элементов восходящего проектирования, *если* программист хорошо чувствует проект своей программы. Новичок или некомпетентный программист может обнаружить, что, когда он реализует верхнюю

и нижнюю части своего проекта, как это показано на рис. 2.14, *б,* на общем уровне эти две составляющие окажутся просто несовместимыми. Если интерфейсы между верхней и нижней частями программы будут совершенно несогласованными, ему, возможно, придется проектировать заново одну или обе эти составляющие. В такой ситуации посредственный программист «догадывается» о необходимости «подогнать» верхнюю часть проекта, изменив ее так, чтобы она была совместима с нижней. Примером тому может служить программа, в которой форматы входных данных выбраны исключительно с целью совмещения верхних уровней программы с некоторыми уже существующими модулями печати отчетов. Это часто приводит к большому замешательству и неудовлетворению пользователя, который находит неприемлемыми форматы как входных данных, так и выходных отчетов!

Тем не менее, если их применять с должной осторожностью, элементы восходящего проектирования могут быть очень полезными. Обычно целесообразно использовать стандартные библиотечные программы; имеет смысл предусмотреть необходимость некоторых стандартных программ ввода-вывода или программ отработки ошибочных ситуаций. Главное, подчеркиваем это здесь, состоит в том, что программист всегда должен осознавать свои действия. Кажется, что многие программисты не осознают, *когда* они пользуются восходящей схемой проектирования. Применяя элементы восходящего проектирования аккуратно и с необходимой осторожностью, программист сможет миновать перечисленные выше опасности.

2.4.3. Нисходящее кодирование и тестирование после нисходящего проектирования

Было замечено, что многие опытные программисты с большим нежеланием применяют на практике идеальную схему нисходящего проектирования; прежде чем приступить к написанию кодов программы, они предпочитают завершить ее проектирование. В обоснование такого подхода чаще всего приводятся следующие доводы.

1. Они просто не верят в идею сочетания нисходящего проектирования, кодирования и тестирования. Хотя эта идея с теоретической точки зрения представляется достаточно разумной, они признают, что предложение — внедрить ее в практику — вызывает у них состояние повышенной нервозности.

2. Они боятся, что сделают ошибку, т. е. приступив к кодированию некоторой части проекта, они могут обнаружить, что в ней допущена ошибка, и им придется заново проектировать (и, следовательно, кодировать) некоторую часть программы.

Мы вполне можем понять первый из этих доводов: люди часто испытывают нервозное состояние, пытаясь освоить что-либо новое, особенно, когда это связано с радикальным отходом от принятых

правил. Второе соображение может вызвать к себе лишь ироническое отношение: именно *потому,* что ошибки возможны, мы и предлагаем придерживаться принципов нисходящей схемы тестирования. Вряд ли что-либо может привести в большее замешательство, чем ситуация, когда проект разработан и утвержден руководством и пользователями, тщательно оформлен и иллюстрирован подробными блок-схемами, программа написана и все испытания проведены, и после этого выясняется, что в действительности пользователь имел в виду нечто совсем иное! Точно так же было бы крайне недальновидным *откладывать* выявление возможных серьезных упущений в проекте до той поры, когда будут написаны все коды программы. Кажется было бы намного разумнее спроектировать *некоторую часть* структуры верхних уровней программы, написать ее коды и провести тестирование с тем, чтобы установить, работоспособна ли общая проектная схема и примут ли ее пользователи.

Тем не менее нужно признать, что в некоторых организациях обсуждаемый подход принят в качестве компромиссной версии нисходящей схемы проектирования. Хотя, быть может, он и не столь изящен, как нам хотелось, это, видимо, некоторый шаг вперед: *после* того как проектирование закончено, мы применяем нисходящее кодирование и тестирование. Таким образом, если в проекте были допущены какие-либо серьезные ошибки, их, возможно, удастся выявить, прежде чем будет закончено кодирование большого числа модулей нижнего уровня.

2.4.4. Нисходящее тестирование после нисходящего проектирования и кодирования

Как мы отметили в предыдущем разделе, многие опытные программисты с большим нежеланием применяют на практике идеальную схему нисходящего проектирования, кодирования и тестирования. Наиболее часто это выражается в том, что сначала программист разрабатывает весь проект программы, а затем приступает к выполнению кодирования и тестирования по нисходящей схеме. Широко распространен также другой вариант, когда проектирование и кодирование по нисходящей схеме осуществляется параллельно; при этомтестированиененачинают до тех пор, пока не завершено написание кодов программы. Когда же приступают к тестированию, то его обычно проводят по принципу «сверху вниз».

Очень часто программисты склоняются к этому варианту по следующим причинам:

1. Они испытывают недостаток машинного времени для проведения тестирования; этот вопрос мы обсудим в разд. 2.4.6.

2. Не отдавая себе в том отчета, они проводят некоторый объем нисходящего тестирования; например, примевяя «настольную» проверку и структурированный разбор на каждом уровне проекта, не

обязательно прибегают при этом к машинной проверке логического скелета программы. Рассматриваемый подход является прогрессивным в сравнении с традиционной восходящей схемой тестирования, и в этом смысле он, по-видимому, не может быть предметом критики. Тот факт, что в этом подходе программист может сознательно или бессознательно выполнять нисходящую «настольную» проверку, также характеризует его как довольно приемлемый на практике.

2.4.5. Проблемы организации и управления

Создается впечатление, что некоторые руководители проектов не пользуются свободой распределения программистов, занятых в проекте, так, как им бы это хотелось; в результате они оказываются *вовлеченными* в процесс разработки, развивающийся вопреки самым лучшим их намерениям по восходящей схеме. Наиболее распространенным тому примером может служить ситуация, когда ответственному руководителю проекта высшим руководством делается замечание: «Пока вы и ведущие разработчики заняты проектированием, рядом сидят без дела десять младших программистов — сделайте с ними что-нибудь!» Пытаясь удержать в своем распоряжении младших исполнителей, руководитель проекта часто придумывает для них какую-нибудь работу, предвидя, возможно, необходимость во многих модулях нижних уровней, которые младшие программисты могут писать, в то время как ведущие программисты заняты проектированием верхних уровней программы.

Как мы уже отмечали, существует опасность, что ко времени объединения в целое верхних уровней структуры с модулями нижнего уровня обнаружится несогласованность соответствующих интерфейсов. Эта опасность обычно возрастает еще в связи с тем, что младшие программисты разрабатывают модули независимо, без должного руководства и достаточного опыта в оценке того, как именнодолжныработатьсоздаваемые ими модули нижнего уровня.

Решение, конечно, следует искать в таком изменении организационной структуры, которое не обязывало бы руководителя проекта привлекать младших исполнителей до того момента, пока в этом нет необходимости; к сожалению, в некоторых фирмах это не так легко осуществить. Положение усугубляется в тех случаях, когда ведущийспециалистпроекта и его руководители томимы предчувствием, что большую часть программной разработки составят многочисленные модули нижнего уровня, которые

1) могут быть написаны младшими программистами без какого-либо наблюдения с их стороны,

2) могут быть написаны до того, как будет окончательно определена структура верхних уровней программы,

3) необходимо начать писать на самых ранних стадиях разработки, чтобы не допустить нарушения сроков.

На самом деле обычно все обстоит как раз наоборот. Модули нижнего уровня *могут* быть написаны младшими программистами, но не в изоляции. Вряд ли есть какой-нибудь смысл в хаотичном написании модулей нижнего уровня, если в дальнейшем их предстоит выбросить из-за несовместимости с верхней структурой программы. И наконец, мало смысла в раннем начале разработки модулей нижнего уровня, если большое число из них окажется бесполезным!

Если руководитель проекта вынужден использовать младших программистов на ранних стадиях проекта (когда, быть может, он предпочел бы их не иметь), он все же имеет возможность включить их в разработку по нисходящей схеме. Например, каждому ведущему программисту можно было бы дать в помощь одного или двух младших программистов; по мере завершения ведущими программистами проектирования верхних уровней младшие программисты могли бы писать соответствующую кодовую реализацию программ этих модулей. В некотором смысле мы подразумеваем, что младшие программисты исполняют роль подручного при ведущем программисте. Грубо говоря, ведущий программист был бы занятболееинтеллектуальными аспектами программирования, а младшие — вопросами исполнения, но при этом они работали бы в одно и то же время над одним и тем же модулем, и это позволило бы осуществить разработку всего проекта по нисходящей схеме.

2.4.6. Недостаток времени для тестирования

Наконец, следует заметить, что некоторым программистам приходится сталкиваться с весьма реальной трудностью, состоящей в нехватке машинного времени для проведения полноценного нисходящего тестирования. Это может происходить по множеству причин, но следующие два представляются наиболее распространенными:

1. Проектирование и кодирование завершено задолго до того, как вычислительная машина попадает в распоряжение программистов. Представляется, *что* это свойственно ряду космических и специальных разработок. То же самое наблюдается в организациях, разрабатывающих ЭВМ, где программисты создают математическое обеспечение для ЭВМ до того, как она фактически изготовлена.

2. Разработка приходится на такое время, когда в организации осуществляется повышенная потребность в машинном времени в связи с другими проектами. Например, если начало разработки приходится на конец года, то планирование этапов тестирования может оказаться очень трудным. Ясно, что нет магических средств решения этих проблем. Тем не менее как программистам, так и руководству следует иметь в виду, что недостаток времени для полноценного тестирования в начале разработки может ощутимо сказы-

ваться на общей стоимости проекта. В настоящее время это, как правило, *не* осознается, а в большинстве организаций считается, что традиционная восходящая схема тестирования вполне удовлетворительна. Если, наконец, будет оценена важность нисходящего тестирования, то это, очевидно, повысит приоритет доступности машинного времени для полноценного тестирования. Среди прочего это, возможно, убедит руководство в том, что необходимо обеспечить доступность ЭВМ для программистов и что работы, завершаемые в конце года, не должны пользоваться абсолютным приоритетом по отношению ко всем остальным видам деятельности.

2.5. Исследования и примеры. Разработка проекта фирмы IBM бригадным методом программирования

К сожалению, многие характерные особенности нисходящейсхемы проектирования порождают сомнения из-за недостатка свидетельств их успешного использования на практике. Например, многие программисты задают вопрос: «Велики ли выгоды, которые можнореально получить, следуя этим правилам и установкам?» Ответ, к сожалению, всегда звучит примерно так: «Ну, я не знаю, но думаю, что они должны быть значительными, а, кроме того, этот подход намного привлекательнее с эстетической точки зрения!»

К счастью, теперь можно сообщить по крайней мере об одном примере разработки, в котором было успешно использовано нисходящее структурное программирование — информационная поисковая система фирмы IBM для газеты *Нью-Йорк Таймс.* Можно надеяться, что к тому времени, когда эта книга будет издана и распространена, появятся другие примеры разработок, которые смогут подтвердить результаты этого проекта. А пока этот единственный пример должен послужить достаточным основанием к тому, чтобы читатель попытался применить некоторые из идей, рассмотренных в этой главе.

Большая часть информации, представленной в этом разделе, заимствована из последнего отчета по проекту для *Нью-Йорк Таймс,* опубликованного в IBM Systems Journal [7]; более подробные сведения можно найти во внутреннем отчете фирмы IBM [8]. Мы дадим краткое описание основных предпосылок и целей этой разработки, опуская детали, не имеющие отношения к содержанию настоящей главы; затем мы сосредоточим наше внимание на методах нисходящего и структурного программирования, использованных IBM. И наконец, что представляется наиболее важным, мы попытаемся показать, насколько удалось фирме IBM добиться эффекта, когда она применила новые методы.

Основные предпосылки разработки. Обращение IBM к идеям структурного программирования стимулировалось результатами более раннего эксперимента этой фирмы по методам разработки с

привлечением суперпрограммистов [9]. Казалось, что фирма IBM интересовалась различными методами, которые позволяли бы проводить разработку проекта малыми группами исполнителей, что можно объяснить естественными преимуществами такого подхода в смысле психологической атмосферы в группе, обмена информацией и стоимости управленческого аппарата. Сначала была опробована идея использования суперпрограммистов: один очень талантливый программист мог бы выполнить работу целой группы из пяти, а то и десяти человек; точно так же малая бригада из трех-пяти суперпрограммистов могла бы взяться за разработку проекта, требующую в других условиях усилий целой «армии муравьев».

В ходе этого эксперимента стало ясным, что время таких программистов является слишком ценным, чтобы его расходовать на множество вспомогательных операций, например перфорацию, написание управляющих карт и подготовку машинных пусков. Кроме того, стало очевидным, что имеется несколько технических аспектов, требующих помощи со стороны *специалистов,* например кого-то, кто досконально знает определенный язык программирования, определенную операционнуюсистему или определеннуюсистемудоступа к файлам. Все эти люди могли бы оказывать помощь суперпрограммисту, с тем чтобы он мог посвятить все свои усилия тем аспектам, в которых он обладает наибольшим талантом,— проектированию и кодированию программ.

Большая часть результатов этого эксперимента оказалась ценной и была использована при разработке проекта для *Нью-Йорк Таймс*. В наименовании новой методологии слово «суперпрограммист» было заменено словами «бригада главного программиста», отражающими представление о такой организации разработки, в которой главному программисту придается группа специалистов и иногда младших программистов. Особенности проекта *Нью-Йорк Таймс* очень интересны, но выходят за рамки обсуждаемых вопросов. Ограничимся здесь лишь краткой цитатой из описания системы [7].

«Ядром информационного банка системы является подсистема диалога, использующая базу данных, состоящую из индексных данных, аннотаций и полных текстов статей из *Нью-Йорк Таймс* и других периодических изданий. Хотя основным назначением системы является формирование файла газетных вырезок для редакционного персонала газеты, система может быть сделана доступной и для удаленных пользователей. Это специализированная система с разделением времени, которая может обслуживать поиск документации для 64 местных терминалов (подсистемыцифровыхтелеэкранных дисплеев IBM 4279/4506) и до 120 линий удаленных экранных или телетайпных терминалов».

Пользователь может просматривать базу данных с помощью тезауруса дескрипторов (ключевых слов), употребленных при иден-

тификации статей. В тезаурусе содержится вся информация о каждом дескрипторе, часто включающая обзорные статьи и требуемые перекрестные ссылки. Имеется возможность отбирать интересующие пользователя дескрипторы и сохранять их для последующего использования при составлении запроса. Опытные пользователи, знающие тезаурус, могут работать непосредственно с дискрипторами. Когда состав перечня дескрипторов определен, задающий вопрос сообщает также следующие библиографические данные, которые дополнительно ограничивают множество интересующих его статей:

- дата или временной интервал дат
- номер газеты, в котором были помещены статьи
- внештатные источники подготовки статьи
- типы статей (например, редакционная или некролог)
- статьи, содержащие специальные виды иллюстраций (например, карты и схемы).

Организация разработки. Как мы уже заметили, фирма IBM изучала возможность организации разработки на основе идеи о главном программисте; на самом деле этот подход был одним из ключевых элементов разработки проекта *Нью-Йорк Таймс*. Если руководитель проекта был ответствен за решение различных финансовых, административных, юридических вопросов и отчетность по проекту, то главный программист отвечал за технические вопросы. Как представляет это автор отчета [7]:

«Принятую организацию разработки можно сравнить с хирургической бригадой, в которой главный программист играет роль, аналогичную роли оперирующего хирурга, опирается на бригаду специалистов (как и хирург), члены которой скорее ассистируют главному специалисту, чем независимо пишут составные части программы. Главный программист — специалист высшей квалификации, отвечающий за детальную разработку программируемой системы. Главный программист целиком пишет важнейшие элементы программируемой системы, а также определяет разбиение системы на подсистемы и способ их объединения. Если система оказывается достаточно целостной по своему назначению и не слишком большой, он может сам написать всю программу».

Библиотека документации проекта. Разработка программы опиралась на развитую систему библиотечных процедур, функционирование которой управлялось и контролировалось опытным библиотекарем. Основное назначение библиотеки состояло в том, чтобы избавить программистов от рутинной работы, а также служить централизованным хранилищем всей документации, листингов, колод исходных программ и других бумажных материалов, связанных с проектом. Говоря точнее, библиотека состояла из следующих четырех разделов:

1. *Внутренняя библиотека,* содержавшая воспринимаемые **машиной исходные программы, переместимые модули, объектные мо-**

дули, операторы редактирования связей, тестовые данные и операторы управления заданиями.

2. *Внешняя библиотека,* содержавшая текущее состояние листингов всех программ, а также листинги последних фиксированных версий программ.

3. Набор *машинных процедур,* который включал всю необходимую информацию для обновления библиотек, выполнения операций по редактированию связей и тестовых пусков, компиляции модулей и хранения объектной программы и обеспечения функционирования библиотек.

4. Набор *формальных процедур,* точно определяющих правила внесения изменений в исходные программы и т. д.; использования машинных процедур; обновления листингов и файлов внешней библиотеки; образования файлов и замены данных в архивах последних версий программ.

Эти процедуры аналогичны процедурам, используемым многими *отделами сопровождения* для управления изменениями в программах после того, как они достигли работоспособного состояния. В подходе IBM заслуживает внимания то обстоятельство, что он использовался с самого начала разработки (хотя это обычно характерно для больших правительственных и военных проектов и *всех* крупных разработок в области ЭВМ). Еще один интересный аспект в использовании библиотеки разработки программы, как это описано в [8], представляет *характер* управления, осуществляемого библиотекарем. Проводится аналогия в функциях библиотекаря и контролера (ответственное лицо финансового отдела) фирмы. Так же как контролер в конечном счете несет ответственность за точность ведения записей финансовых операций фирмы (на самом деле на контролера часто официально возложена ответственность за любые ошибки в записях и поэтому он, как правило, полномочен запретить другим служащим фирмы — вплоть до президента — несанкционированный доступ или внесение изменений), библиотекарь в конечном счете отвечает за точность и сохранность записей, сопровождающих разработку. Отсюда следует, что библиотекарь может запретить программистам, включая главного программиста, вносить поправки, проводить тестовые пуски или любые другие действия, не выполнив стандартные библиотечные процедуры.

Структурное и нисходящее программирование. Структурное программирование и идеи нисходящего программирования, рассмотренные в этой главе и гл. 4, составили важные элементы проекта IBM. Проектируя системы, программисты прежде всего записывали операторы языка управления заданиями, требующиеся для компиляции, загрузки и исполнения необходимых модулей системы. Затем в состав этой структуры были введены фиктивные реализации модулей и файлов, чтобы обеспечить тестирование верхних уровней системы, в то же время было начато нисходящее проекти-

рование действительных реализаций модулей. Хотя других подробностей не сообщалось, представляется, что этот подход оказался успешным [7]:

«Успешно применено также нисходящее программирование. Логическая структура одной из важнейших программ оказалась правильной в первом же пуске и не претерпела никаких изменений в последовавшем процессе расширения программы до окончательных размеров. Это облегчило отладку, поскольку обычно программы пускались в комплексе и редкие ошибки легко выявились анализом вновь реализованных функций системы. Нисходящим программированием упростились также проблемы интерфейса, обычно свойственные разработкам с участием многих программистов, так как интерфейсы всегда определялись и кодировались раньше каких-либо функциональных программ, использующих зти интерфейсы».

Принципы структурного программирования в этом проекте совпадали стеми, что описываются в гл. 4; в качестве основного языка программирования использовался язык ПЛ/1, и для организации передач управления программистам разрешалось использовать только операторы IF-THEN-ELSE и DO-WHILE. Кроме того, были предприняты энергичные усилия, направленные на использование осмысленных идентификаторов и ограничение максимальной длины модулей исходной программы приблизительно 50 операторами. Для большего удобства в осуществлении структурного программирования программистам было разрешено использовать следующие дополнительные возможности: моделируемый оператор CASE алгольного типа (сопровождаемый оператором GO TO и массивом переменных типа метка) и итеративное применение оператора DO с описателем WHILE или без него.

Результаты проекта фирмы IBM. С точки зрения исследователей в области ЭВМ, наибольшую ценность проекта фирмы IBM составляет то, что в нем были проведены измерения производительности программистов с целью оценки полезности примененных в разработке принципов управления, библиотеки разработки программ и идей структурного программирования.

В табл. 2.1 приведено количество написанных в течение разработки программных команд; и табл. 2.1, итабл. 2.2 взяты из работы [7]. Важно отметить, что эта разработка не относится к числу *малых;* она не уступает многим техническим проектам средних размеров. В другой таблице из работы [7], которая здесь не приводится, показано число человеко-месяцев, израсходованных различными участниками разработки. С учетом руководителя проекта и секретаря на всю разработку потребовалось 132 человеко-месяца, т. е. приблизительно столько же, сколько требуется для программной разработки средних размеров в экономических и деловых проектах.

Таблица 2.1

РАСПРЕДЕЛЕНИЕ ЧИСЛА **СТРОК ПРОГРАММНОГО КОДА** ИСХОДНОЙ ПРОГРАММЫ ПО СЛОЖНОСТИ И УРОВНЮ ЯЗЫКА ПРОГРАММИРОВАНИЯ

	Уровень язык.		
Сложность	высокий	низкий	всего
Сложные	5034		5034
Стандартные	44247	4513	48760
Простые	27897	1633	29530
Всею	77178	6146	83324

При составлении этой таблицы использовались следующие соглашения:

Простая программа характеризуется малым числом взаимосвязей с другими элементами системы. (Большая часть вспомогательных программ относилась к этой категории.)

Программа стандартной сложности характеризуется заметным числом взаимосвязей с другими элементами системы. (Примерами могут служить функциональные модули подсистемы обеспечения диалога и система редактирования данных при вводе.)

Сложные программы отличаются большим числом взаимосвязей с другими элементами системы. (К этой категории относятся управляющие модули системы обеспечения диалога.)

Программы высокого уровня кодирования включают коды, написанные на таких языках, как ПЛ/1, Кобол или JCL.

Программы низкого уровня кодирования включают программы, написанные на языке ассемблера, а также управляющие операторы редактирования связей.

В табл. 2.2 отражены производительности программистов, измеренные несколькими различными способами. При учете трудозатрат только программистов в проекте в среднем создавалось по 65 отлаженных команд на программиста в день — это приблизительно в четыре-шесть раз лучше того, что может достичь средний программист. Даже с учетом сотрудников, занятых в различных формальных, управленческих и вспомогательных операциях, среднее число отлаженных команд на человека в день оказалось равным 35, и этот показатель значительно выше, чем в большинстве программирующих организаций. Отметим также, что, хотя введение в состав групп библиотекарей программ и необходимого штата по формальным операциям, возможно, и снизило суммарную производительность персонала, работа этого дополнительного штата обычно обходится значительно дешевле (в смысле расхода на зарплату и других сопутствующих выгод), чем работа программистов.

Не ясно, в какой мере успех проекта обусловлен выбранной организацией разработки и освобождением программистов от рутин-

Таблица 2.2

ПРОИЗВОДИТЕЛЬНОСТЬПРОГРАММИСТОВВПРОЕКТЕ ФИРМЫ IBM

Состав	Число строк исходной программы на человека в день
Проектирование, программирозание, отладка и тестирование этдельных элементов программы	65
Z учетом всех специалистов	47
Z учетом библиотекаря	43
Вся бригада	35

ных операций, а в какой мере — применением структурного программирования и нисходящего проектирования. Наибольшее внимание в отчете [7] уделяется вопросам *организации разработки,* и это можно понять. Хотя и отмечено, что идеи структурного программирования имели важное значение, не удалось как-либо определить *меру* их полезности. Тем не менее успех, достигнутый в разработке этого проекта, мог бы служить для других достаточным стимулом, чтобы попытаться его повторить.

ЛИТЕРАТУРА

1. Wulf W. A., A Case Against the GOTO, Proceedings of the 25th National ACM Conference, v. 2, p. 791—797.
2 Wirth N., Program Development by Stepwise Refinement, Communications of the ACM, April 1971.
3. Wirth N., Systematic Programming, Prentice-Hall, Inc., 1971; имеется русский перевод: Вирт Н., Систематическое программирование, изд-во «Мир», М., 1977.
4. Constantine L. L., Concepts in Program Design, Information and Systems Press, Cambridge, Massachusetts, 1967.
5. Yourdon E., Call 360 Costs, Datamation, Nov. 1 1971.
6. StevensW. P., Myers G. J., Constantine L. L., Structured Design, IBM Systems Journal, May 1974.
7. Baker F. T., Chief Programming Team Management of Production Programming, IBM Systems Journal, Jan. 1972, p. 56—73.
8. Baker F. T., Chief Programmer Teams: Principles and Procedures, Report N FSC 71-5108, IBM, Federal Systems Division, Gaithersburg, Maryland 20760.
9. Aron J. D., The Superprogrammer Project, Software Engineering Techniques, NATO Scientific Affairs Division, Brussels 39, Belgium, p. 50—52.
10. Mills H., Top-Down Programming in Large Systems из DebuggingTechniques in LargeSystems, Prentice-Hall, Inc., 1971; имеется русский перевод: Средства отладки больших систем, под ред. Р. Растина, изд-во «Статистика» М., 1977.

ВОПРОСЫ

1. Требуется ли в вашей организации соблюдение стандартной формы построения программы типа представленной на рис. 2.1? Обязаны ли все программисты пользоваться такой формой? Находите ли вы эти требования разумными?

2. Если вы работаете в области экономических прикладных задач, то считаете ли вы разумным предполагать, что все ваши программы должны иметь форму, показанную на ри£. 2.2? Можете ли вы представить себе какие-нибудь приложения, для которых эта форма неприменима?

3. Дайте краткое определение нисходящего проектирования. В чем назначение этого принципа?

4. Пользуетесь ли вы и другие программисты вашей организации нисходящей схемой проектирования? Если нет, то почему? Считаете ли вы, что если бы ваше руководство ввело эту схему проектирования программ как обязательную, то трудно было бы вам подчиниться такому требованию?

5. Назовите три вызывающих осложнения обстоятельства, с которыми программисты могут столкнуться пытаясь проектировать программы по нисходящей схеме. Думаете ли вы, что можете лично столкнуться с этими трудностями?

6. Почему важно пользоваться формализованными и точными определениями входных данных, функций и выходных данных каждого модуля программы, проектируемой по нисходящей схеме? Существуют ли какие-нибудь методы, облегчающие получение таких формализованных определений?

7. Почему важно избегать обсуждения мелких деталей в процессе проектирования программы? Каким образом можно установить, что процесс проектирования начинает увязать в мелочах? Как можно избежать такой ситуации?

8. В разд. 2.1.3 было рекомендовано на каждом уровне проекта записывать программную реализацию модуля в виде не более чем одной страницы кодов. Какова цель этой рекомендации? Что можно было бы сделать в случае, если программу нельзя представить в виде одной страницы простейших операторов?

9. В разд. 2.1.3 говорилось о недостаточной формальности Кобола при определении интерфейсов между различными модулями программы. Почему это так? Является ли это, на ваш взгляд, важным недостатком? Что можно в связи с этим предпринять? Являются ли другие языки, например Фортран и ПЛ/1, столь же неформальными?

10. В чем состоит достоинство систем идентификации структуры программы, подобных методологии HIPO фирмы IBM?

11. Почему, на ваш взгляд, пытаясь разбить программу на небольшие модули, программисты делают это столь небрежно? Происходит ли это из-за недостатка опыта в подобной работе или потому, что, начиная проектирование программы, они не вполне отчетливо его мысленно себе представляют? Может быть, имеются еще какие-нибудь причины?

12. В какой степени, на ваш взгляд, структура проекта вашей программы отражает структуру выданных вам требований? То есть не пришли вы к выводу, что, когда вам выдают плохо организованные требования, это часто приводит к плохо организованной программе? Что можно в связи с этим предпринять? Кто в большей степени повинен в этом: системный аналитик или программист?

13. Назовите три причины, по которым системные аналитики иногда составляют плохо организованные, «неструктурированные» требования. Считаете ли вы, что эти причины часто возникают в вашей организации? Осознают ли их системные аналитики? А руководство?

14. Считаете ли вы, что системных аналитиков следует обучать некоторым навыкам в нисходящем проектировании? Возникли бы при этом какие-нибудь трудности в вашей организации? Противились бы этой идее системные аналитики?

15. Дайте краткое определение нисходящего кодирования. В чем назначение этого принципа?

16. Рассмотрите три положительных качества нисходящего кодирования. Считаете ли вы эти качества важными? Уверены ли вы сами в реальности этих преимуществ? Как считают другие сотрудники вашей организации?

17. Почему блок-схемы оказываются иногда недостаточным средством передачи проектных решений между программистами? Предпочитаете ли вы пользоваться блок-схемами?

18. Почему программные коды могут служить более точным средством передачи идей по проекту между программистами? С какими трудностями может столкнуться применение этого средства?

19. Каким образом, на ваш взгляд, следует располагать коды в листинге программы? Предпочитаете ли вы горизонтальную организацию, показанную на рис. 2.8, б, или вертикальную, представленную рис. 2.9, в? В чем преимущества и недостатки каждой из этих организаций?

20. Дайте краткое определение нисходящего тестирования. В чем назначение этой концепции? В чем разница между нисходящей и восходящей схемами тестирования?

21. Какой методологии тестирования программ следуют в вашей организации? Стандартизированы ли процедуры тестирования? Если допускается восходящее тестирование, то по *каким причинам* оно используется? Найдите кого-нибудь, кто предпочитает эту схему, и попросите обосновать ее использование. Имеется ли какой-нибудь смысл в этих доводах?

22. Одним из основных принципов нисходящего тестирования является использование фиктивных модулей и программных «болванок» вместо нереализованных разделов программного кода. Приведите пример программы, в которой можно было бы употребить каждый из следующих *типов* фиктивных модулей:

а) Фиктивный модуль, сразу отдающий управление на выход.

б) Фиктивный модуль, вырабатывающий постоянное значение выходных данных.

в) Фиктивный модуль, вырабатывающий случайное значение выходных данных.

г) Фиктивный модуль, выдающий на печать отладочное сообщение.

д) Фиктивный модуль, реализующий упрощенный вариант окончательной версии программы.

23. В разд. 2.3.1 дается пример плана нисходящего тестирования программы по рис. 2.11. Дайте описание подобного плана тестирования программы, над которой вы работаете в настоящее время или которую вы недавно закончили.

24. В качестве одного из первых шагов нисходящего тестирования рекомендуется написать для программы карты управления заданиями и прогнать фиктивную версию всей программы, чтобы убедиться в правильности этих управляющих карт. Считаете ли вы этот шаг разумным? На ваш взгляд, следует ли от каждого требовать выполнения этого шага?

25. Перечислите пять положительных особенностей нисходящего тестирования. Какое из этих достоинств вы считаете наиболее важным? Какое из них наименее важное?

26. Почему тестирование по нисходящей схеме позволяет нам исключить (в соответствии с принятым определением) испытания на уровне системы? Считаете ли вы это важным обстоятельством? Пожелали бы, на ваш взгляд, пользователи и руководство вашей организации упразднить испытания на уровне системы?

27. В чем состоит преимущество максимально раннего испытания важнейших интерфейсов программы? Считаете ли вы, что для больших проектов это важнее, чем для малых?

28. В чем преимущество ознакомления пользователя с предварительной версией программы на ранней стадии нисходящего тестирования? Считаете ли вы, что это может быть важным достоинством нисходящей схемы тестирования? Можете ли вы назвать какие-нибудь неудобства, заключенные в таком предложении?

29. Часто ли в вашей организации работы по программированию регламентируются нереалистичными сроками? Как часто ваши программные разработки отстают от планового графика? Считаете ли вы, что в таких случаях было бы полезным иметь законченную часть программы, доступную для восприятия применением нисходящей схемы тестирования?

30. Почему в нисходящем тестировании отладка может быть проще, чем в восходящем? Считаете ли вы, что она может быть *существенно* проще?

31. В качестве эксперимента подыщите приятеля или коллегу приблизительно равной с вашей квалификацией в программировании. Возьмите задачу на программирование, которой оба вы могли бы заняться. Тестируйте ваш вариант решения по нисходящей схеме, а партнеру предложите применить восходящую схему. Когда станет ясно, что оба варианта испытаны в равной степени, дайте ответ на следующие вопросы:

а) Тестирование какой из программ потребовало больше суммарного времени?

б) Тестирование какой из программ потребовало больше времени ЦП?

32. Почему применение нисходящей схемы тестирования может улучшить психологическую атмосферу. Может ли это иметь большое значение для малых программных разработок? Для больших?

33. Используется ли в вашей организации моделирование тестовой обстановки, описанное в разд. 2.3.2? Считаете ли вы, что такой подход может быть заменен применением нисходящего тестирования?

34. Какие варианты нисходящего тестирования на практике применяются? С какими ситуациями в программировании связано такое применение? Можно ли, на ваш взгляд, установить правила, точно определяющие ту модификацию нисходящего тестирования, которую необходимо использовать?

35. Назовите четыре причины, по которым использование восходящего тестирования иногда оказывается необходимым. Считаете ли вы, что подобные ситуации наблюдаются часто? Можно ли их избежать?

36. Приведите пример программы, в которой в рамках нисходящей схемы формирование тестовых данных для полноценного испытания модуля нижнего уровня было бы затруднительным. Часто ли возникают подобные ситуации в вашей организации? Как следует поступать, когда это случается?

37. В чем состоит назначение структурированных разборов? Считаете ли вы, что это — хорошая идея? Может ли она быть одобрена другими программистами, с которыми вы работаете? Если бы руководство вашей организации ввело обязательное для всех разработок применение структурированных разборов, то трудно было бы вам подчиниться этому требованию? Какие осложнения вы могли бы предвидеть в связи с этим подходом?

38. Почему руководитель проекта не допускается к участию в структурированных разборах? Считаете ли вы, что это — важная мера предосторожности? На ваш взгляд, согласился бы ваш руководитель с этим условием?

39. Почему в ходе структурированных разборов стремятся не столько предложить новый проект, сколько обнаружить пороки в существующем?

40. Каковы основные доводы программистов против структурированных разборов?

41. Опишите такую ситуацию в программировании, в которой требовалось бы применить сочетание нисходящей и восходящей схем проектирования. Часто ли может возникнуть такая ситуация? Требуется ли в этом случае от программиста какое-нибудь специальное планирование?

42. Почему некоторые программисты настаивают на завершении этапа проектирования (по нисходящей схеме), прежде чем начать кодирование и тестирование (по такой же схеме)? Считаете ли вы их доводы основательными? К. каким затруднениям может привести такой подход?

ГЛАВА 3

МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ

3.0. Введение

Модульность в программировании подобна честности в политике: каждый утверждает, что она — одно из его достоинств, но кажется, *никто* не знает, что она собой представляет, как ее привить, обрести или добиться. Каждый младший программист знает, что модульная программа — это такая программа, в которой любую часть логической структуры можно изменить, не вызывая изменений в остальных частях программы. Однако, что же это на самом деле означает? Каким образом программист узнает, что он написал модульную программу? Может быть это — особое свойство ума, некая утонченная форма кармы? Важнее понять, каким образом узнает руководитель, что программист написал модульную программу? Существует ли какая-нибудь мера модульности, т. е. могу ли я сказать, что моя программа в 4,3 раза модульнее вашей?

Годами эти вопросы беспокоили программистов и ученых — специалистов по ЭВМ. Если бы мы *знали,* как писать модульные программы, то в течение нескольких последних лет каждый писал бы только такие программы. Те программисты, которые упорно сопротивлялись бы этому, были бы преданы анафеме и навсегда дисквалифицированы. Проблемы отладки, модификации и сопровождения программ упростились бы на порядок, а число разводов и язвенных заболеваний среди программистов резко бы сократилось.

К сожалению, этот утопический век еще не настал, хотя имеется некоторая надежда, что он *наступит* в недалеком будущем. Лишь совсем недавно начала проясняться формализованная теория модульности, и эта теория обычно формулируется на языке структурного программирования, которое мы рассмотрим в следующей главе. Хотя оно таит в себе большие преимущества в сравнении с практическими специальными методами достижения модульности, структурное программирование еще не стало общепринятой идеологией. По разным причинам программисты, кажется, предпочитают пользоваться своими индивидуальными методами обеспечения модульности. Не очень удивляет то, что хорошие программисты, как

правило, успешно пользуются своими особыми методами, а посредственные программисты обычно создают программы, степень модульности которых оказываются значительно ниже планировавшейся.

Мы надеемся, что структурное программирование постепенно станет общепринятым стандартным методом построения модульных программ. До той поры, однако, остаются в ходу существующие и все еще полезные специальные несистематизированные методы, которые заслуживают рассмотрения и будут обсуждены в этой главе. Сначала мы попытаемся дать определение модульности. Затем обсудим некоторые «за» и «против» модульных программ. Интересно в связи с этим рассмотреть некоторые возражения, выдвигаемые теми программистами, которые все еще упорно остаются в оппозиции к общим принципам модульного программирования[1]). В заключение, и это наиболее важный раздел, мы обсудим несколько полезных приемов создания модульных программ.

3.1. Определение модульности

3.1.1. Размеры модулей в модульной программе

Так как все модульные программы по предположению состоят из модулей, один из наиболее интересных частных способов определения *модульности* опирается на определение самого модуля. В своих попытках определить характерные черты модульных программ специалисты предлагают следующие стандарты размеров модуля:

1. Некоторые пользователи и программисты фирмы IBM предлагают определить модуль как нечто, укладывающееся в 4096 байтах памяти.

2. Аналогично некоторые, программирующие для систем фирмы Honeywell UNIVAC и других, предлагают считать модулем всякую последовательность кодов, занимающую 512 слов памяти, или 1024 слова, или 2048 слов и т. д.

3. Некоторые руководители отделов программирования высказывали пожелания, чтобы модуль определялся как раздел программы, который может быть написан и отлажен одним программистом за один месяц. Считая приближенно, что средний программистможетнаписатьот Юдо 15отлаженных командвдень, получим отсюда размер модуля равным 200—300 программным командам.

[1]) В декабре 1972 г. я читал курс программирования повышенной сложности в одном из правительственных агентств в Калифорнии. Во время обсуждения модульного программирования один из старших программистов сделал следующее замечание: «Я программирую на Коболе уже более пяти лет и никогда еще не пользовался оператором PERFORM. Не вижу причин, по которым я должен бы начать делать это теперь».

4. Мой коллега Л. Константайн высказал предположение, что большинство хороших модульных программ не содержит модулей, превышающих 100—200 комачд.

5. Другой мой коллега, Р. Хенри, предположил, что модуль не должен состоять более чем из 20 операторов на языке высокого уровня. У него такое чувство, что если программист записывает что-то с помощью более чем 20 операторов, то скорее всего он программирует не один функциональный элемент, а в этом случае следует употреблять больше одного модуля.

6. Сотрудник фирмы IBM Бейкер в своей статье (Baker F. T. «Chief Programmer Team Management of Production Programming», IBM Systems Journal, v. 11, №2; более подробно об этой статье см.вгл. 4) рекомендуетсвоимпрограммистамограничиватьразмеры модулей 50 операторами на языке ПЛ/1.

7. В 1970 г. автор познакомился с одной большой разработкой для ВВС США, в которой предполагалось объединить несколько независимых систем управления запасами и материально-технического обеспечения. Объединенная система должна была иметь базу данных с непосредственной связью емкостью около 22 млрд. символов и содержать прикладных программ общей длиной около миллиона строк кодов на Коболе. Предполагалось, что она будет создана усилиями приблизительно пятисот программистов. Имея дело с такой грандиозной задачей, руководители проекта определили стандарт, по которому ни одному программисту не разрешалось писать модули, превышающие 500 операторов Кобола.

Должно быть ясным, что наложение ограничений на размер программы не гарантирует того, что она будет модульной. Один из программистов проекта для ВВС США, описанного выше, сделал интересное замечание по поводу правила о 500 Кобол-операторах в модуле: «Черт с ними, я соглашусь со всеми глупыми ограничениями вроде этого, большинство из них можно довольно просто обойти. Предлагаемое правило модульности слишком просто. Я пишу программу, не обращая никакого внимания на это правило 500 операторов, а затем, если окончательная программа получается длиной в 3000 операторов, я просто делаю: чик! чик! чик! — получаю шесть модулей! Вот и все!».

С другой стороны, приведенные выше положения не лишены определенного смысла. Один руководитель из числа моих друзей сказал как-то: «Слушай, я знаю, что модульная программадолжна поддаваться разбиению на несколько подпрограмм, но у меня нет времени, чтобы убедиться в том, что программисты обеспечивают это должным образом. Они способны принести мне главную программу, состоящую из 500 операторов, к которой добавлены два-три модуля по 10 операторов, и всю эту чепуху назвать модульной программой! Единственное средство, которое может меня убедить в том, что они правильно представляют себе саму *идею* модульности

и правильно применяют это понятие, состоит в том, чтобы определить жесткий стандарт в 50 операторов на модуль. Если у них появляется необходимость в каком-либо исключении, они сначала должны разъяснить мне его суть».

3.1.2. Независимость

Одним из более абстрактных понятий, связанных с модульностью, является понятие *независимости:* в модульной программе каждый модуль (например, каждая подпрограмма, каждая секция в Кобол-программе, каждая ПЛ/1-процедура) не зависит от других. Под этим, конечно, подразумевается, что его можно изменить или модифицировать, не вызывая каких-либо последствий в других модулях. Должно быть ясным, однако, что понятие независимости не является абсолютным. Нам было бы очень трудно сказать, что модуль X на 99 или 44% независимее всех других модулей (если бы нам это удалось, то, может быть, в конечном счете мы сказали бы, что программа Z модульна на 99 или 44%).

С другой стороны, может иметь смысл представление о независимости по отношению к другим факторам, например к базе данных, объему рабочей памяти, другим модулям и т. д. На самом деле мы хотим задать следующий вопрос: если изменить один фактор в программе, то как это повлияет на данный модуль? Так, по-видимому, разумно рассматривать независимость некоторого модуля (и в конечном счете его модульность) по отношению к таким факторам, как:

1. Логическая структура программы, т. е. алгоритм. Если вся программа (или система) зависит от некоторого специального подхода (например, в задаче обхода сети построение программы зависит от выбора эвристического алгоритма обхода сети), то в скольких модулях потребуется внести изменения при изменении алгоритма?

2. Аргументы, или параметры, модуля. В этом отношении зависимость модуля может быть довольно сильной: если изменяется число, тип или формат аргументов, то не следует слишком удивляться, если это потребует больших изменений в модуле.

3. Внутренние переменные таблиц и константы. Многие модули зависят от общих таблиц (например, от разделов протоколов и сообщений в ряде экономических систем и систем коммутации сообщений); если изменяется структура таких таблиц, то мы можем ожидать, что модули также изменятся.

4. Структура и формат базы данных. В большей степени эта зависимость аналогична зависимости от общих «переменных и таблиц, упомянутой выше, с той разницей, что с практической точки зрения базу данных удобнее считать независимой от программы.

Интересно заметить, что в таких разработках, как CODASYL, одна из основных задач группы разработки базы данных состояла в таком определении базы данных, при котором программа от нее совершенно не зависит.

5. Модульная структура управления программой. Некоторые пишут модуль не особенно задумываясь над тем, каким образом он будет использоваться. Допустим, что, после того как мы записали модуль X, мы вдруг узнаем, что необходимо обеспечить его реентерабельность. Какую *часть* логической структуры модуля нам придется изменить? Допустим, что модуль входит в систему реального времени и первоначально программист посчитал, что, работая, модуль не будет прерываться. Какую часть модуля пришлось бы изменить, если бы оказалось, что в действительности его работа *может* прерываться? Какие бы возникли трудности, если бы вдруг нам захотелось использовать модуль рекурсивно?

Предполагая эти факторы неизменными, мы могли бы установить независимость отдельных модулей программы. Если интерфейсы между модулями определены, то в этом случае можно было бы заменять модуль функционально ему эквивалентным (т. е. таким, который принимает те же самые входные данные и вырабатывает те же значения выходных данных), не вызывая при этом никаких последствий ни в каком другом модуле программы. В зависимости от того, насколько возможны такие замены, имеет смысл говорить о том, что мы имеем модульную программу.

Заметим, что определенное таким образом свойство независимости нарушается, если некоторые модули могут произвольно передавать или получать управление от одного к другому или если они могут модифицировать друг друга. Таким образом, следующее условие модульности и составляющая часть независимости — условие «один вход — один выход», т. е. *модульная программа должна состоять из модулей, которые имеют одну точку входа и одну точку выхода*. Мы можем несколько ослабить это требование, допуская, быть может, существование модулей с более чем одним входом; важно при этом, что точки входов должны быть строго определены и другие модули не могут входить в данный в произвольной точке. Помимо этого, мы настаиваем на том, чтобы данный модуль не мог непосредственно изменять команды другого модуля, например, с помощью оператора ALTER в Кобол-программах.

3.2. Преимущества и недостатки модульности

Прежде чем обсудить некоторые методы написания модульных программ, мы должны привести некоторые доводы в пользу модульности, а также ряд контрдоводов.

3.2.1. Доводы в пользу модульности

Мы уже упоминали в этой главе достоинства модульности; здесь достаточно было бы просто их повторить. Тем не менее всякому программисту следует подумать о каждом из нижеследующих пунктов в связи с его собственной программой — просто, чтобы убедиться, что они *имеют* к ней отношение.

1. Модульные программы легко составлять и отлаживать. Функциональные компоненты такой программы могут быть написаны и отлажены порознь.

2. Модульную программу легче сопровождать и модифицировать. Функциональные компоненты могут быть изменены, переписаны или заменены без изменений в остальных частях.

3. Руководству легче управлять разработкой модульной программы. Более сложные модули могут быть переданы более опытным программистам; простые модули могут быть написаны младшими программистами. Разбивая программу на модули, которые могут быть созданы за один месяц, руководитель может быть уверен, что ни один из программистов не окажется слишком перегруженным сложными элементами программы.

3.2.2. Доводы против модульности

Напомнив многие достоинства модульности, мы можем теперь поставить вопрос: почему же многие программы не отличаются большей степенью модульности. С уверенностью можно сказать, что большинство программ пишется *немодульными*. Многие крупные фирмы расходуют до 50% *бюджета, выделенного на обработку данных,* на сопровождение, модификацию и совершенствование программ, и большую часть этих действий можно было бы выполнять совершенно тривиально, если бы с самого начала проектировалась модульная программа.

Большинство программистов не понимают смысла модульности. Вероятно, основная причина недостаточной модульности программ состоит в том, что большинство программистов в действительности не знают, что такое модульность. У них есть интуитивное представление о значении этого слова и методах достижения модульности, но, по-видимому, они не задумывались над формализованными теоремами и алгоритмами. Не многие программисты согласятся, что именно в этом заключена действительная причина недостаточной модульности их программ; они обычно приводят какие-нибудь другие доводы из числа перечисленных ниже. Больше всего озадачивает то, что они представляют себе модульность *абсолютным* понятием и что, вводя несколько подпрограмм, они смогут ее обеспечить.

Модульность требует большей дополнительной работы. Чтобы писать модульные программы, программист должен быть значи-

тельно более аккуратным на этапе проектирования программной разработки. Он должен проектировать свои программы по нисходящей схеме (рассмотренной в гл. 2), начиная с верхних уровней всей программы (системы) и затем продвигаясь вниз к более детальному проектированию отдельных подпрограмм. На каждом шаге он должен спрашивать себя, легко ли будет внести изменения в проект и просто ли его модифицировать. В больших разработках каждый шаг проектирования должен сопровождаться составлением соответствующей документации, с тем чтобы различные пользователи и руководство могли выразить свое понимание и одобрение проводимой работы.

Все это требует огромного терпения и заметного объема довольно кропотливой и усердной работы, и все это — до начала написания программы. Во многих случаях программист оказывается в условиях (часто им же и созданных!) необходимости писать программу как можно скорее, и он может отказаться от этой дополнительной работы.

К сожалению, существующие методы решения этой проблемы не вполне удовлетворительны. Руководитель, имеющий дело с непокладистым программистом, может просто *приказать* емуписать программу модульно. В иных случаях руководитель может попытаться убедить его, показав, что модульность позволит в дальнейшем облегчить изменение программы. Однако часто руководителю очень трудно *доказать,* что модульный подход определенно облегчит модификации программ; во всяком случае, это может не заинтересовать программиста, поскольку он полагает, что за модификацию программы будет отвечать кто-нибудь другой.

Модульный подход иногда требует большего времени ЦП. Эта проблема возникает прежде всего в тех случаях, когда программа отличается наличием большого числа подпрограмм, написанных на языках высокого уровня. Кобол, Фортран и ПЛ/1 упоминаются в этой связи в первую очередь; так, в одной версии ПЛ/1 требуется 198 мкс только на то, чтобы войти в подпрограмму и выйти из нее! Для обеспечения модульности может также потребоваться больше времени ЦП, если части программы с командами ввода-вывода совершенно отделены от ее вычислительных частей; входная запись может быть несколько раз передана подпрограммам, прежде чем начнется ее фактическая обработка.

Если этот вопрос может оказаться критическим, программисту (или его руководителю следует сделать оценку, позволяющую установить, действительно ли требование модульности *существенно* увеличивает время прохождения программы. Жесткие правила, безусловно запрещающие программистам пользоваться операторами вызова подпрограмм (которые часто встречаются в стандартных учебниках Кобола для некоторых малых ЭВМ), вообще говоря, заслуживают порицания. В большинстве случаев модульный под-

ход требует дополнительно 5—10% времени ЦП; представляется, что это приемлемая плата за возможность легко изменять программу; исключение составляют очень специальные случаи (например, некоторые прикладные системы реального времени или программы, расходующие по нескольку часов машинного времени).

В модульном подходе может потребоваться несколько больший объем памяти. Если каждой подпрограмме отводится отдельная часть рабочей памяти, то всей программе может потребоваться несколько больший объем памяти; однако это не обязательно так, если промежуточные результаты хранятся в списке, располагаемом в памяти магазинного типа (исключение составляют случаи вызова многократно вложенных подпрограмм). В тех случаях, когда программа реализуется на основе.большого числа подпрограмм, для обеспечения связей между ними также может потребоваться дополнительная память.

И в этом случае программисту следует получить надежную оценку того, что модульный подход требует *существенно* большего объема памяти. В большинстве случаев модульность не приводит к увеличению объема программ более чем на 5—10%.Это не может приводить к каким-либо осложнениям, за исключением случаев, когда на длину программы накладывается произвольное ограничение, или при использовании мини-ЭВМ из-за чисто физических ограничений.

Модульность может оказаться причиной трудностей в системах реального времени и системах типа онлайн. Во многих вычислительных системах реального времени и системах типа онлайн недостаточно просто разбить программу на несколько небольших, логически законченных программ так, чтобы они в каждый момент времени согласованно делили память. Программист должен также позаботиться о том, чтобы определенные программы оказывались в оперативной памяти в определенное время; в противном случае программа будет переводиться в режим «ожидание» всякий раз, когда необходимо переслать в оперативную память новую подпрограмму.

Эти трудности могут быть еще более серьезными в системах с виртуальной памятью, например в новых машинах Системы IBM/370. Программист в этом случае должен не только позаботиться о разбиении его программы на подпрограммы, но. также и о том, чтобы подпрограммы согласованно располагались в пределах страницы памяти машины. Подпрограммы, которые нередко вызывают одна другую (часто упоминаемые как «рабочий набор» — термин, введенный в обращение проф. Деннингом), должны располагаться либо на одних и тех же страницах, либо на множестве страниц, которые обязательно занимают физическую память в одно и то же время.

3.3. Методы построения модульных программ

В вводном разделе этой главы было сделано предварительное замечание о том, что законченная теория модульности не является общепринятой и все еще разрабатывается. Тем не менее существует некоторое число специальных приемов, которые обычно позволяют успешно строить модульные программы; некоторые из них будут изучены в этом разделе.

3.3.1. Разбивайте ваши программы на малые независимые подпрограммы

Как мы отметили ранее, одна из наиболее распространенных формулировок определяет модульную программу как такую, которая составлена из модулей, имеющих длину около 50 операторов, или 20 операторов, или 100 операторов и т. д., т. е. кому что больше нравится. В связи с этим как самое главное следует иметь в виду то, что это требование является только *средством,* но не *самоцелью;* разбиение программы на модули объемом в 50 операторов ни в коей мере не гарантирует, что программа будет обладать свойством модульности.

Другим определяющим качеством в этом методе является *независимость:* если подпрограммы сильно зависят одна от другой, то их введение более чем бесполезно (так как на них будет расходоваться дополнительное время ЦП и дополнительная память). Необходимо постоянно стремиться выделить *функциональный* характер подпрограммы; каждая подпрограмма должна иметь определенное назначение, которое более или менее не зависит от других подпрограмм программы. Конечно, это не всегда просто сделать — небрежное мышление и проектирование часто приводят к программе, представляющей собой хотя и единую, но весьма большую сеть взаимосвязанных подпрограмм, в которой сравнительно малые изменения в одной из подпрограмм вызовут, вероятно, реакцию во всех остальных.

Выбор определенного размера «идеального» модуля ни в коей мере нельзя назвать наукой. Необходимо учитывать используемый язык, характер применения и квалификацию как программиста-разработчика, так и программиста последующего сопровождения. В некоторых организациях выбор в качестве стандартного размера модуля фрагмента в 50 операторов объясняется просто ощущением, что такой модуль может быть размещен на одной странице листинга программы. Распространено представление, что средний программист в состоянии прочесть и понять смысл лишь такого модуля, который не переходит на следующую страницу листинга. В других случаях (например, в программах на языке ассемблера)может ока-

заться целесообразным выбирать модули большего или меньшего размера.

Здесь уместно высказать еще одно пожелание: смотрите, не *переусердствуйте* в модульности. Редко есть какой-нибудь смысл разбивать программу на модули из трех-четырех операторов каждый; такие попытки, наверное, приведут к большому увеличению времени ЦП, а также необходимого объема памяти.

3.3.2. Используйте таблицы решений

Изучая теорию переключательных схем в электротехнике, мы обычно выделяем два типа элементов: *комбинационные элементы,* значение выходных параметров которых зависит *только* от условий на входе в тот же момент времени, и *последовательностные* элементы, значение выходов которых есть функция текущих условий на входе и состояния элемента в тот же момент времени (которое в свою очередь зависит от предшествующих условий на входе). Большое число программ для ЭВМ может быть охарактеризовано точно таким же способом: действия, выполняемые программой, зависят от нескольких достаточно сложно взаимосвязанных условий, часть из которых может быть отнесена к исходным условиям, а часть — может быть определена динамически в процессе функционирования программы.

С этой точки зрения сравнение теории переключательных схем с программированием для ЭВМ представляется весьма интригующим. Существует развитая теория и практические методы, позволяющие инженеру строить модульные коммутационные сети минимальной сложности (хотя соответствующая теория представляется более точной в случае комбинационных элементов). В программировании для достижения тех же целей используется ряд методов, произвольно именуемых «таблицы решений». В последующем мы увидим, что эти методы обладают рядом очень важных достоинств:

1. Они способствуют лучшему анализу и пониманию задачи.
2. Они представляют более четкое средство общения между программистом и пользователем, исключающее многие неоднозначности, свойственные словесным описаниям программируемой задачи.
3. Они обеспечивают более широкий контроль ошибок, исключая возможные неполноту, противоречивость и избыточность.

Кроме того, программа построенная на основе таблиц решений, оказывается достаточно модульной, что является причиной обсуждения этих методов в настоящем разделе.

Если программа очень простая, включающая лишь немногие условия, то ее анализ, проектирование и реализация часто легко осуществляются на основе лишь интуитивных представлений; при этом вопросам модульности часто не придают большого значения,

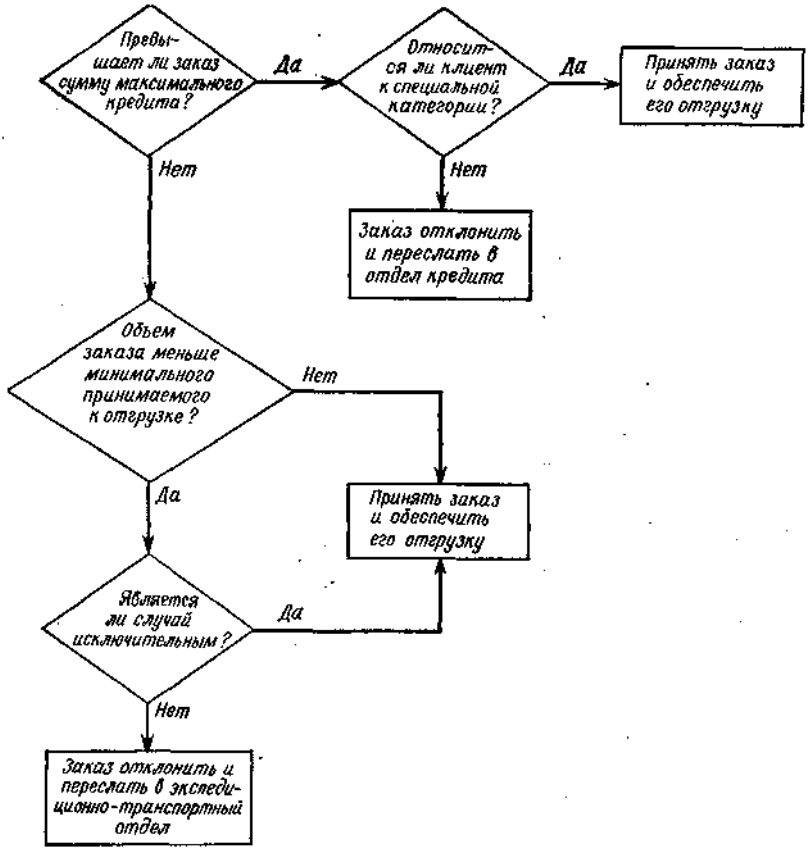


Рис. 3.1. Блок-схема типичной логической программы.

поскольку всю программу можно быстро переписать. Однако при увеличении числа входных данных и усложнении взаимосвязей между ними интуитивные методы могут оказаться «бессистемными» и «немодульными». Рассмотрим следующий простой пример (для которого в реальных условиях не потребовалось бы использования формализованных методов на основе таблиц решений, причем и сам по себе пример слишком тривиален, чтобы отвечать каким бы то ни было реальным условиям!).

Если клиент делает заказ, превышающий максимальную сумму выделяемого ему кредита, то заказ пересылается в отдел кредитов. Заказ, однако, должен быть принят, если клиент относится к специальной категории, т. е. является одним из тех, кто регулярно пользуется нашими услугами. К тому же если объем заказа меньше минимального, принимаемого к отгрузке, то заказ отклоняется и пересылается руководителю отдела отгрузки. Система, однако,

должна обладать способностью исключать это правило в том случае, если клиент будет настаивать на отгрузке, несмотря на слишком малый объем заказа.

Взявшись за эту задачу, средний программист мог бы воспользоваться множеством подходов, один из которых представлен на рис. 3.1. После ознакомления со спецификациями задачи первое его побуждение — нарисовать блок-схему (мы не рассматриваем тех немногочисленных программистов, первое побуждение которых — сесть за перфоратор и начать кодирование программы!). После того как он убедит себя в том, что блок-схема соответствует требованиям, он начнет кодирование программы, а чтобы она была модульной, он, вероятно, использует как можно большее число подпрограмм!

В подобных поспешных подходах имеется несколько недостатков. Прежде всего должно быть очевидным, что *требования,* представляющие, по-видимому, типичный пример требований, составленных неспециалистом в области вычислительной науки, сами по себе довольно нечетки и немодульны. Создается впечатление, что они следуют ходу мыслей пользователя, когда он перебирает условие за условием, пытаясь припомнить свои обычные действия при поступлении заявок на исполнение заказов. Приведенные требования не дают указаний о действиях программы в том случае, когда стоимость получаемого заказа превышает максимальный кредит, но при этом его объем ниже минимального, принимаемого к отгрузке; такое сочетание условий, может быть, не пришло на ум пользователю, когда он составлял требования, поскольку оно возникает только раз в полгода. Конечно ясно, что если требования неполны или противоречивы, то и блок-схема рис. 3.1 будет неполной или противоречивой; программная реализация этой блок-схемы может только увенчать эту глупую ситуацию.

Помимо того, в данной ситуации пользователь, безусловно, пожелает изменить требования (поскольку исходные не годятся!), что приведет к необходимости нарисовать заново блок-схему и, следовательно, заново кодировать программу. Заметим, что обилие подпрограмм при этом мало помогает,— незначительные изменения в требованиях могут вызвать существенную перестройку блок-схемы, что привело бы к изменению всей логической структуры программы.

Сравнительно лучшим подходом к решению задачи является построение *таблицы решений.* Мы можем начать с перечисления четырех возможных *условий* на входе:

1. Значение стоимости в долларах превышает максимальный кредит.
2. Клиент имеет специальное разрешение отдела кредита.
3. Объем заказа меньше минимального объема, принимаемого к отгрузке.

Условия и действия	Правила															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
1. Значение стоимости заказа в долларах превышает максимальный кредит	F	F	F	F	F	F	F	F	**T**	T	**T**	**T**	**T**	T	**T**	**T**
2. Клиент пользуется специальным разрешением отдела кредита	F	F	F	F	T	T	T	**T**	F	F	F	F	**T**	**T**	**T**	**T**
3. Объем заказа меньше минимально допустимого	F	F	T	T	F	F	T	**T**	F	F	T	T	F	F	**T**	**T**
4. Клиент пользуется специальными льготами экспедиционно-транспортного отдела	F	T	F	T	F	T	F	**T**	F	T	F	**T**	F	T	F	**T**
1. Отклонить заказ, направить в отдел кредита									X	X	X	X				
2. Отклонить заказ, направить в экспедиционно-транспортный отдел			X				X				X				X	
3. Принять груз и обеспечить его отгрузку	X	X		X	X	X		X					X	X		X

Заметим, что в 11-й ситуации заказ отклоняется дважды. Есть ли в этом какое-нибудь удобство или хотя бы смысл? Для простоты в дальнейшем мы будем полагать необходимым только действие № 1

Рис. 3.2. Простая таблица решений. F — ложно, T — истинно.

4. Транспортный отдел подтвердил прием заказа к исполнению. Так же мы можем установить, что существуют три *действия,* которые может выполнить программа:

1. Заказ не исполняется и пересылается в отдел кредита.

Условия и действия	Правила															
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
. Значение стоимости заказа в долларах превышает максимальный кредит	T	**T**	T	**T**	F	F	**T**	F	F	F	F	F	F	**T**	**T**	**T**
2. Клиент пользуется специальным разрешением отдела кредита	F	F	F	F	F	T	**T**	F	F	F	T	T	T	**T**	T	**T**
3. Объем заказа меньше минимально допустимого	F	F	T	T	T	**T**	**T**	F	F	T	F	F	T	F	F	**T**
4. Клиент пользуется специальными льготами экспедиционно-транспортного отдела	F	T	F	**T**	F	F	F	F	T	**T**	F	T	**T**	F	T	**T**
1. Отклонить заказ, направить в отдел кредита	X	X	X	X												
2. Отклонить заказ, направить в экспедиционно-транспортный отдел					X	X	X									
3. Принять заказ и обеспечить его отгрузку								X	X	X	X	X	X	X	X	X

В третьем правиле заметим, что мы произвольно решили исполнять только действие №1 (см. правило № 11 на рис. 3.2)

Рис. 3.3. Преобразованная таблица решений.

2. Заказ не исполняется и пересылается в экспедиционно-транспортный отдел.

3. Заказ исполняется нормально.

Таблица решений, или истинностная таблица, строится для того, чтобы можно было видеть все комбинации входных и выходных условий. В нашем примере возможны $2^4=16$ *комбинаций* условий, как это видно из таблицы на рис. 3.2. Эта таблица решений

```
C   ПРИ ВХОДЕ В ЭТОТ РАЗДЕЛ ПРОГРАММЫ ЦЕЛОЕ К УКАЗЫВАЕТ,
C   В КОТОРОЙ ИЗ 16 ВОЗМОЖНЫХ СИТУАЦИЙ МЫ НАХОДИМСЯ.
C   ВЫЧИСЛЯЕМЫЙ ОПЕРАТОР GO TO ПЕРЕДАЕТ УПРАВЛЕНИЕ
C   СООТВЕТСТВУЮЩЕМУ РАЗДЕЛУ ПРОГРАММЫ ДЛЯ ОБРАБОТКИ
C   ЭТОГО СЛУЧАЯ. ЗАМЕТЬТЕ, ЧТО СУЩЕСТВУЕТ ТОЛЬКО ТРИ ТА-
C   КИХ РАЗДЕЛА ПРОГРАММЫ—ПО ОДНОМУ НА КАЖДОЕ ИЗ ТРЕХ
C   ВОЗМОЖНЫХ ДЕЙСТВИЙ
    GOTO (3, 3, 2, 3, 3, 3, 2, 3, 1, 1, 1, 1, 3, 3, 2, 3), K
C
C   ЭТО ПРОГРАММА ОБРАБОТКИ ТРЕТЬЕГО ТИПА ДЕЙСТВИЙ
C   НОРМАЛЬНОЕ ИСПОЛНЕНИЕ ЗАКАЗА
C
3   PRINT 100, ACCTNO, PARTNO, QTY
```

Рис. 3.4.а. Реализация таблицы решений с использованием вычисляемых операторов GO TO.

```
C     ПРИ ВХОДЕ В ЭТОТ РАЗДЕЛПРОГРАММЫ МЫ ПРЕДПОЛАГАЕМ,
C     ЧТО РАНЕЕ БЫЛИ ОПРЕДЕЛЕНЫ СЛЕДУЮЩИЕ ПЕРЕМЕННЫЕ:
C       REAL DOLLAR СТОИМОСТЬ ЗАКАЗА КЛИЕНТА В ДОЛЛАРАХ
C       REAL CREDIT МАКСИМАЛЬНЫЙ КРЕДИТ ЭТОГО КЛИЕНТА
C       REAL ORDRSZ ОБЪЕМ ЗАКАЗА ЭТОГО КЛИЕНТА
C       REAL ORDRMN МИНИМАЛЬНЫЙ ОБЪЕМ, ПРИНИМАЕМЫЙ К ОТ-
C                   ГРУЗКЕ
C     МЫ ПРЕДПОЛАГАЕМ ТАКЖЕ, ЧТО
C       SPCRED = 0    ЕСЛИ ЭТОТ КЛИЕНТ НЕ ИМЕЕТ СПЕЦИ-
C                      АЛЬНОГО РАЗРЕШЕНИЯ ОТДЕЛА КРЕДИТА
C       SPCRED=1      ЕСЛИЭТОТКЛИЕНТИМЕЕТСПЕЦИАЛЬНОЕ
C                     РАЗРЕШЕНИЕ ОТДЕЛА КРЕДИТА
C       SPSHIP = 0    ЕСЛИЭТОТКЛИЕНТНЕИМЕЕТСПЕЦИАЛЬНОГО
C             .       РАЗРЕШЕНИЯ ЭКСПЕД.-ТРАНСПОРТНОГО ОТ-
C                     ДЕЛА
C       SPSHIP=1      ЕСЛИЭТОТ КЛИЕНТ ИМЕЕТ СПЕЦИАЛЬНОЕ
                      РАЗРЕШЕНИЕ ЭКСПЕД.-ТРАНСПОРТНОГО ОТ-
                      ДЕЛА
      K = 1
      IF (DOLLAR .GE. CREDIT) K = K + 8
      IF (SPCRED .EQ. 1) K = K + 4
      IF (ORDRSZ .LE. ORDRMN) K = K + 2
      IF (SPSHIP .EQ. 1) K = K + 1
      GO TO (3, 3, 2, 3, 3, 3, 2, 3, 1, 1, 1, 1, 3, 3, 2, S), K
```

Рис. 3.4б. Реализация таблицы решений на Фортране.

```
/* ПРИ ВХОДЕ В ЭТОТ РАЗДЕЛ ПРОГРАММЫ МЫ ПРЕДПОЛАГАЕМ,
ЧТО В СООТВЕТСТВУЮЩЕМ МЕСТЕ БЫЛИ ОПИСАНЫ СЛЕДУЮЩИЕ
ПЕРЕМЕННЫЕ, А ТАКЖЕ, ЧТО БЫЛО ОБЕСПЕЧЕНО СЛЕДУЮЩЕЕ
ИХ НАЧАЛЬНОЕ СОДЕРЖАНИЕ:
    DOLLAR СТОИМОСТЬ ЗАКАЗА КЛИЕНТА В ДОЛЛАРАХ
    CREDIT МАКСИМАЛЬНЫЙ КРЕДИТ ЭТОГО КЛИЕНТА
    ORDRMN МИНИМАЛЬНЫЙ ОБЪЕМ ЗАКАЗА, ПРИНИМАЕМЫЙ.
           К ОТГРУЗКЕ
    ORDRSZ ОБЪЕМ ЗАКАЗА ЭТОГО КЛИЕНТА
МЫ ТАКЖЕ ПРЕДПОЛАГАЕМ, ЧТО ПЕРЕМЕННЫЕ "SPCRED" И "SPSHIP"
УЖЕ БЫЛИ ОПИСАНЫ КАК СТРОКИ БИТОВ ДЛИНЫ (1), А ИХ НА-
ЧАЛЬНОЕ СОДЕРЖАНИЕ БЫЛО ЗАДАНО СЛЕДУЮЩИМ ОБРАЗОМ:
    SPCRED = 0 ЕСЛИ ЭТОТ КЛИЕНТ НЕ ИМЕЕТ СПЕЦИАЛЬНОГО
               РАЗРЕШЕНИЯ ОТДЕЛА КРЕДИТА
           = 1 ЕСЛИ ЭТОТ КЛИЕНТ ИМЕЕТ СПЕЦИАЛЬНОЕ РАЗРЕ-
               ШЕНИЕ ОТДЕЛА КРЕДИТА
    SPSHIP = 0 ЕСЛИ ЭТОТ КЛИЕНТ НЕ ИМЕЕТ СПЕЦИАЛЬНОГО
               РАЗРЕШЕНИЯ ЭКСПЕД.-ТРАНСПОРТНОГО ОТДЕЛА
           = 1 ЕСЛИ ЭТОТ КЛИЕНТ ИМЕЕТ СПЕЦИАЛЬНОЕ РАЗРЕ-
               ШЕНИЕ ЭКСПЕД.-ТРАНСПОРТНОГО ОТДЕЛА
ЗАМЕТЬТЕ, ЧТО ЭТУ ПРОГРАММУ МОЖНО НАПИСАТЬ МНОЖЕСТВОМ
ДРУГИХ СПОСОБОВ, ИСПОЛЬЗУЯ ГИБКИЕ И ШИРОКИЕ ВОЗМОЖНОСТИ
ПЛ/1*/
DECLARE K FIXED BINARY;
DECLARE TABLE(16) LABEL lNITIAL(PROCESS,PROCESS,REJECTSHIP,
   PROCESS, PROCESS, PROCESS, REJECTSHIP, PROCESS, REJECT-
   CREDIT, REJECTCREDIT, REJECTCREDIT, REJECTCREDIT, PROCESS,
   PROCESS, REJECTCTSHIP, PROCESS);
K=1;
IF DOLLAR > CREDIT THEN K = K + 8;
IF SPCRED THEN K = K + 4;
IF ORDRSZ < ORDRMN THEN K = K + 2;
IF SPSHIP THEN K = K + 1 :
GO TO TABLE(K);
```

Рис. 3,4в. Реализация таблицы решений на ПЛ/1.

может быть преобразована так, чтобы все комбинации входных условий, ведущих к действию 1, объединялись в одну группу и т.д.; эта форма таблицы решений показана на рис. 3.3; опираясь на такую таблицу решений, часто удается разработать оптимальную по эффективности программу.

Отметим, что таблицы рис. 3.2 и 3.3 носят исчерпывающий характер, т. е. в них перечисляются *все возможные комбинации* входных условий, хотя многие из этих комбинаций оказываются невозможными, неприемлемыми, избыточными или бессмысленными.

```
NOTE ПОЯСНИТЕЛЬНЫЕ ЗАМЕЧАНИЯ. ПРИ ВХОДЕ В ЭТОТ ФРАГМЕНТ
     ПРОГРАММЫ МЫ ПРЕДПОЛАГАЕМ, ЧТО РАНЕЕ БЫЛИ ОПРЕ-
     ДЕЛЕНЫ ПЕРЕЧИСЛЯЕМЫЕ НИЖЕ ПЕРЕМЕННЫЕ И ЧТО ИМ
     БЫЛИ ПРИСВОЕНЫ СЛЕДУЮЩИЕ НАЧАЛЬНЫЕ ЗНАЧЕНИЯ:
          DOLLAR  СТОИМОСТЬ ЗАКАЗА КЛИЕНТА В ДОЛЛАРАХ
          CREDIT  МАКСИМАЛЬНЫЙ КРЕДИТ ЭТОГО КЛИЕНТА
          ORDRMN  МИНИМАЛЬНЫЙ ОБЪЕМ ЗАКАЗА, ПРИНИМАЕ-
                  МЫЙ К ДОСТАВКЕ
          ORDRSZ  ОБЪЕМ ЗАКАЗА ЭТОГО КЛИЕНТА
     МЫ ПРЕДПОЛАГАЕМ ТАКЖЕ, ЧТО РАНЕЕ БЫЛИ УСТАНОВЛЕНЫ
     ДВА ПРИЗНАКА:
          SPCRED = 0  ЕСЛИ ЭТОТ КЛИЕНТ НЕ ИМЕЕТ СПЕЦИАЛЬ-
                      НОГО РАЗРЕШЕНИЯ ОТДЕЛА КРЕДИТА
                 = 1  ЕСЛИ ЭТОТ КЛИЕНТ ИМЕЕТ СПЕЦИАЛЬНОЕ
                      РАЗРЕШЕНИЕ ОТДЕЛА КРЕДИТА
          SPSHIP = 0  ЕСЛИ ЭТОТ КЛИЕНТ НЕ ИМЕЕТ СПЕЦИАЛЬ-
                      НОГО РАЗРЕШЕНИЯ ЭКСПЕД.-ТРАНСПОРТ-
                      НОГО ОТДЕЛА
                «=1   ЕСЛИ ЭТОТ КЛИЕНТ ИМЕЕТ СПЕЦИАЛЬНОЕ
                      РАЗРЕШЕНИЕ ЭКСПЕД.-ТРАНСПОРТНОГО ОТ-
                      ДЕЛА
ТАБЛИЦА РЕШЕНИЙ. MOVE ONE TO K.
IF DOLLAR IS GREATER THAN CREDIT THEN ADD EIGHT TO K.
IF SPCRED IS EQUAL TO ONE THEN ADD FOUR TO K.
IF ORDRSZ IS LESS THAN ORDRMN THEN ADD TWO TO K.
IF SPSHIP IS EQUAL TO ONE THEN ADD ONE TO K.
GO TO PROCESS, PROCESS, REJECTSHIP, PROCESS, PROCESS, PROCESS,
     REJECTSHIP, PROCESS, REJECTCREDIT, REJECTCREDIT,
     REJECTCREDIT, REJECTCREDIT, PROCESS, PROCESS,
     REJECTSHIP, PROCESS DEPENDING ON K.
```

Рис. 3.4г. Реализация таблицы решений на Коболе.

Несмотря на это, таблицы решений обладают тем достоинством, что приводят к очень простой программной реализации: использованием приема «индексируемой передачи управления», иллюстрируемого в общем виде рис. 3.4а. На рис. 3.4б приводится кодовая реализация рассматриваемого примера, записанная на Фортране с применением вычисляемого оператора GO TO; рис. 3.4г иллюстрирует тот же прием в Коболе с использованием оператора GO TO DEPENDING ON; рис. 3.4в, 3.4д и 3.4е показывают решение той же самой задачи, записанное на языке ПЛ/1, языках ассемблера Системы IBM/360 и PDP-10.

Удобный в случае небольшого числа условий (не превышающего, например, пяти или шести) метод исчерпывающих таблиц становится весьма громоздким, когда число условий возрастает.

```
* ПРИ ВХОДЕ В ЭТОТ РАЗДЕЛ ПРОГРАММЫ МЫ ПРЕДПОЛАГАЕМ,
* ЧТО СЛЕДУЮЩИЕ ПЕРЕМЕННЫЕ ОПИСАНЫ И ИМ ПРИСВОЕНЫ
* НАЧАЛЬНЫЕ ЗНАЧЕНИЯ:
*       DOLLAR                  СТОИМОСТЬ ЗАКАЗА КЛИЕНТА В ДОЛ-
*                               ЛАРАХ
*       CREDIT                  МАКСИМАЛЬНЫЙ КРЕДИТ ЭТОГО
*                               КЛИЕНТА
*       ORDRSZ                  ОБЪЕМ ЗАКАЗА ЭТОГО КЛИЕНТА
*       ORDRMN                  МИНИМАЛЬНЫЙ ОБЪЕМ ЗАКАЗА, ПРИ-
*                               НИМАЕМЫЙ К ОТГРУЗКЕ
* МЫ ТАКЖЕ ПРЕДПОЛАГАЕМ, ЧТО ПЕРЕМЕННЫЕ "SPCRED" И
* "SPSHIP" ОПРЕДЕЛЕНЫ КАК ОДНОБИТОВЫЕ ПРИЗНАКИ СО СЛЕ-
* ДУЮЩИМИ ЗНАЧЕНИЯМИ:
*       SPCRED = 0              ЕСЛИ ЭТОТ КЛИЕНТ НЕ ИМЕЕТ СПЕ-
*                               ЦИАЛЬНОГО РАЗРЕШЕНИЯ ОТДЕЛА
*                               КРЕДИТА
*              =1               ЕСЛИЭТОТКЛИЕНТИМЕЕТСПЕЦИАЛЬ-
*                               НОЕ РАЗРЕШЕНИЕ ОТДЕЛА КРЕДИТА
*       SPSHIP  =0              ЕСЛИ ЭТОТ КЛИЕНТ НЕ ИМЕЕТ СПЕ-
*                               ЦИАЛЬНОГО РАЗРЕШЕНИЯ ЭКСПЕД.-
*                               ТРАНСПОРТНОГО ОТДЕЛА
*              =1               ЕСЛИ ЭТОТ КЛИЕНТ ИМЕЕТ СПЕЦИ-
*                               АЛЬНОЕ РАЗРЕШЕНИЕ ЭКСПЕД.-ТРАН-
*                               СПОРТНОГО ОТДЕЛА
BEGIN   BALR  R15,0             ЗАДАТЬАДРЕСАПЕРЕМЕЩАЕМОЙПРО-
                                ГРАММЫ
        USING*,  R15
        L       R1,ZERO         ЗАДАНИЕ НАЧАЛЬНОГО ЗНАЧЕНИЯ
                                ИНДЕКСА
        L       R2,  CREDIT
        C       R2,  DOLLAR     БОЛЬШЕ ЛИ CREDIT, ЧЕМ DOLLAR?
        BH      TEST2           ЕСЛИ ДА, ТО ОБОЙТИ СЛЕДУЮЩУЮ
                                КОМАНДУ
        A       R1,  EIGHT      ИНАЧЕ УВЕЛИЧИТЬ ЗНАЧЕНИЕ ИН-
                                ДЕКСА НА 8
TEST2   CLI     SPCRED,  0      ЕСТЬ ЛИ У КЛИЕНТА СПЕЦИАЛЬНОЕ
                                РАЗРЕШЕНИЕ?
        BE      TEST3           ЕСЛИНЕТ,ТООБОЙДИСЛЕДУЮЩУЮ
                                КОМАНДУ
        A       R1,  FOUR       ИНАЧЕ УВЕЛИЧИТЬ ЗНАЧЕНИЕ ИН-
                                ДЕКСА НА 4
TEST3   L       R2,  ORDRMN
        C       R2,  ORDRMN     МЕНЬШЕ ЛИ ORDRMN, ЧЕМ ORDRSZ?
        BL      TEST4           ЕСЛИ ДА, ТО ОБОЙТИ СЛЕДУЮЩУЮ
                                КОМАНДУ
        A       R1,  TWO        ИНАЧЕ УВЕЛИЧИТЬ ЗНАЧЕНИЕ ИН-
```

```
                                ДЕКСА НА 2
TEST4  CLI    SPSHIP, 0        ЕСТЬ ЛИ У КЛИЕНТА СПЕЦИАЛЬНОЕ
                                РАЗРЕШЕНИЕ?
       BE     DONE             ЕСЛИ НЕТ, ТО ОБОЙТИ СЛЕДУЮЩУЮ
                                КОМАНДУ
       A      R1,ONE           ИНАЧЕ УВЕЛИЧИТЬ ЗНАЧЕНИЕ ИН-
                                ДЕКСА НА 1
DONE   IC     R1,TABLE(R1)     ПОСЛАТЬЗНАЧЕНИЕИНДЕКСАВВЕК-
                                ТОР ПЕРЕДАЧИ УПРАВЛЕНИЯ
       B      VECTOR-4 (R1)    ПЕРЕДАТЬ УПРАВЛЕНИЕ СООТВЕТ-
                                СТВУЮЩЕЙ ПОДПРОГРАММЕ
*
* ТАБЛИЦА АДРЕСОВ ВЕКТОРА ПЕРЕДАЧИ УПРАВЛЕНИЯ
*
TABLE  DC                       XOC, OC, 08, 0C, 0C, 0C, 08, 0C, 04, 04,
                                04, 04, 0C, 0C, 08, 0C'
*
* ВЕКТОР ПЕРЕДАЧИ УПРАВЛЕНИЯ
*
VECTOR B REJECTC                ОТКЛОНИТЬ ЗАКАЗ И ПЕРЕСЛАТЬ
                                В ОТДЕЛ КРЕДИТА
       B REJECTS                ОТКЛОНИТЬ ЗАКАЗ И ПЕРЕСЛАТЬ
                                В ОТДЕЛ ДОСТАВКИ
       B PROCESS                ИСПОЛНИТЬ ЗАКАЗ НОРМАЛЬНО
EIGHT  DC    F'8'
FOUR   DC    F'4'
TWO    DC    F'2'
ONE    DC    F'1'
ZERO   DC    F'0'
R1     EQU   1
R2     EQU   2
R3     EQU   3
```

Рис. 3.4д. Реализация таблицы решений на языке ассемблера Системы IBM/360.

Если бы нам пришлось учитывать 15 условий, то в нашем вычисляемом операторе GO TO было бы 32768 адресов! В подобных случаях можно было бы прибегнуть к построению «селективной» таблицы решений, в которой учитываются лишь определенные «интересные» комбинации условий. Селективная таблица решений для нашего примера условий приема заказа показана на рис. 3.5.

Представление решения вформеталицы, показанной на рис. 3.5, может выглядеть эквивалентным исходной блок-схеме, изображенной на рис. 3.1; однако теперь, когда мы уже проделали эту процедуру составления таблицы решений, у нас появилась большая уверенность в том, что блок-схема рис. 3.1 действительно соответ-

```
; ПРИ ВХОДЕ В ЭТОТ РАЗДЕЛ ПРОГРАММЫ МЫ ПРЕДПОЛАГАЕМ,
; ЧТО СЛЕДУЮЩИЕ ПЕРЕМЕННЫЕ ОПИСАНЫ И ИХ НАЧАЛЬНЫЕ
; ЗНАЧЕНИЯ ЗАДАНЫ:
; DOLLAR      СТОИМОСТЬ ЗАКАЗА КЛИЕНТА В ДОЛЛАРАХ
; CREDIT      МАКСИМАЛЬНЫЙ КРЕДИТ ЭТОГО КЛИЕНТА
; ORDRSZ      ОБЪЕМ ЗАКАЗА ЭТОГО КЛИЕНТА
;    ORDRMN   МИНИМАЛЬНЫЙ ОБЪЕМ ЗАКАЗА, ПРИНИМАЕМЫЙ
              К ОТГРУЗКЕ
; МЫ ТАКЖЕ ПРЕДПОЛАГАЕМ, ЧТО ПЕРЕМЕННЫЕ "SPCRED" И
;"SPSHIP" ОПРЕДЕЛЕНЫ И ЗАДАНЫ СЛЕДУЮЩИМИ НАЧАЛЬНЫМИ
; ЗНАЧЕНИЯМИ:
;    SPCRED = 0 ЕСЛИ ЭТОТ КЛИЕНТ НЕ ИМЕЕТ СПЕЦИАЛЬНОГО
;     РАЗРЕШЕНИЯ ОТДЕЛА КРЕДИТА
;  = 1    ЕСЛИ ЭТОТ КЛИЕНТ ИМЕЕТ СПЕЦИАЛЬНОЕ
           РАЗРЕШЕНИЕ ОТДЕЛА КРЕДИТА
;    SPSHIP =0 ЕСЛИ ЭТОТ КЛИЕНТ НЕ ИМЕЕТ СПЕЦИАЛЬНОГО
           РАЗРЕШЕНИЯ ЭКСПЕД.-ТРАНСПОРТНОГО ОТДЕЛА
;          = 1 ЕСЛИ ЭТОТ КЛИЕНТ ИМЕЕТ СПЕЦИАЛЬНОЕ
           РАЗРЕШЕНИЕ ЭКСПЕД.-ТРАНСПОРТНОГО ОТДЕЛА
    MOVEI  1,0          ЗАДАНИЕ НАЧАЛЬНОГО ЗНАЧЕНИЯ ИН-
                        ДЕКСА
    MOVE   2, CREDIT    БОЛЬШЕ ЛИ CREDIT, ЧЕМ DOLLAR?
    CAMG   2, DOLLAR    ЕСЛИ ДА, ТО ОБОЙТИ СЛЕДУЮЩУЮ
                        КОМАНДУ
    ADDI   1, 10        ИНАЧЕ УВЕЛИЧИТЬ ИНДЕКС НА 8
    SKIPE  SPCRED       ЕСТЬ ЛИ У КЛИЕНТА СПЕЦИАЛЬНОЕ РАЗ-
                        РЕШЕНИЕ?
    ADDI   1,4          ЕСЛИ ДА, ТО УВЕЛИЧИТЬ ИНДЕКС НА 4
    MOVE   2, ORDRMN    МЕНЬШЕ ЛИ ORDRMN, ЧЕМ ORDRSZ?
    CAML   2, ORDRSZ    ЕСЛИ ДА, ТО ОБОЙТИ СЛЕДУЮЩУЮ
                        КОМАНДУ
    ADDI   1,2          ИНАЧЕ УВЕЛИЧИТЬ ИНДЕКС НА 2
    SKIPE  SPSHIP       ЕСТЬ ЛИ У КЛИЕНТА СПЕЦИАЛЬНОЕ
                        РАЗРЕШЕНИЕ?
    ADDI   1,1          ЕСЛИ ДА, ТО УВЕЛИЧИТЬ ИНДЕКС НА 1
    JRST   @TABLE (1) ПЕРЕЙТИ К СООТВЕТСТВУЮЩЕЙ ПОДПРО-
                        ГРАММЕ

; ТАБЛИЦА ПЕРЕДАЧ УПРАВЛЕНИЙ ДЛЯ 16 ВОЗМОЖНЫХ СЛУЧАЕВ

TABLE: EXP PROCESS, PROCESS, REJECTS, PROCESS
       EXP PROCESS, PROCESS, REJECTS, PROCESS
       EXP REJECTC, REJECTC, REJECTC, REJECTC
       EXP PROCESS, PROCESS, REJECTS, PROCESS
```

Рис. 3.4е. Реализация таблицы решений на языке ассемблера PDP-10.

Условия и действия	Правила					
	1	2	3	4	5	6
1. Стоимость заказа в долларах превышает максимальный кредит	T	—	F	T	F	T
2. Клиент пользуется специальным разрешением отдела кредита	F	—	—	T	—	T
3. Объем заказа меньше минимально допустимого	—	T	F	F	T	T
4. Клиент пользуется специальным разрешением экспедиционно-транспортного отдела	—	F	—	—	T	T
1. Отклонить заказ, направить в отдел кредита	X					
2. Отклонить заказ, направить в экспедиционно-транспортный отдел		X				
3. Принять заказ и обеспечить отгрузку			X	X	X	X

Заметьте, что правила № 1 и 2 до некоторой степени перекрываются: ситуация, отмеченная на рис. 3.2 № 11, удовлетворяет обоим правилам № 1 и 2. Однако, поскольку *сначала* мы будем проверять применимость правила № 1 (см. рис. 3.6а), эта селективная таблица вполне эквивалентна таблице рис. 3.3.

Рис. 3.5. Селективная таблица решений для примера о приеме заказов.

ствует выбраннымтребованиям. В этом, вероятно, состоит наибольшая ценность таблиц решений: они вынуждают программиста сначала организовать свои мысли и выразить их в табличной форме и лишь *затем* приступить к изображению блок-схемы. К тому же должно быть очевидным, что метод таблиц решений может способствовать построению более модульной программы.Этов особенности справедливо в отношении исчерпывающей таблицы решений, показанной на рис. 3.2: если пользователь изменит свое мнение относительно действий, которые необходимо выполнять при некоторой определенной комбинации условий, то нам достаточно будет лишь изменить соответствующий адрес входа в вычисляемом операторе GO TO представленных иллюстративных программ.

Некоторые программисты предпочитают преобразовать исходную форму таблицы решений, показанную на рис. 3.2, в форму, изображенную на рис. 3.3. Соответствующие блок-схемы принимают вид, показанный на рис. 3.6а и рис. 3.6б. Этому решению также свойственна определенная модульность: если бы пользователь решил изменить *характер* действия № 1 (например, вместо отказа в исполнении заказа и пересылки его в отдел кредита он мог бы решить отпечатать уведомление об отказе с отсылкой одной копии

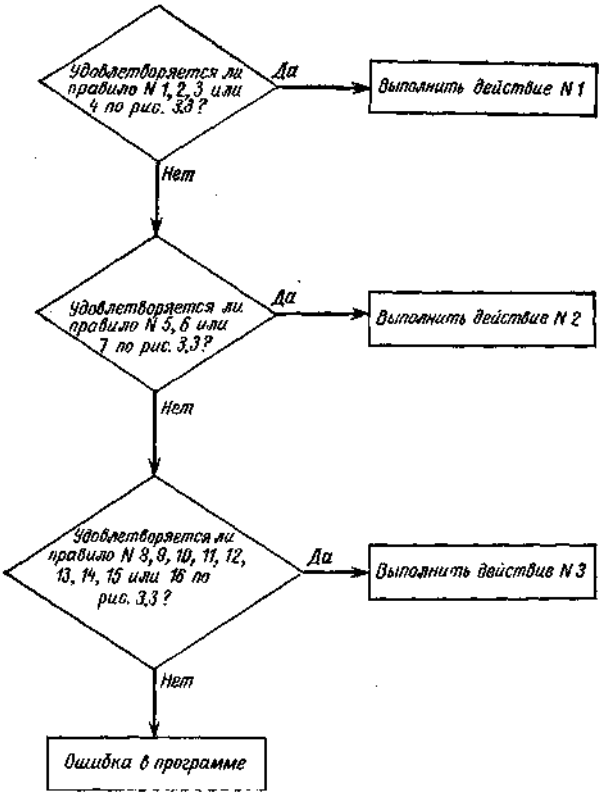


Рис. 3.6а. Блок-схема, построенная на основе преобразованной таблицы решений.

в отдел кредита, одной копии — клиенту и одной копии — отделу сбыта), то реализовать это решение в блок-схеме было бы очень просто. С другой стороны, если бы пользователь пожелал изменить условия, которые влекут за собой определенное действие, то блок-схемы, показанные на рис. 3.6, пришлось бы рисовать целиком заново.

Если мы несколько продолжим эту мысль, то увидим, что решение в форме селективной таблицы, показанной на рис. 3.5, может быть вообще лишено явных признаков модульности. Если пользователь изменит требования, то придется строить новые правила отбора, что может привести к необходимости переписать таблицу решений; это в свою очередь повлекло бы за собой выполнение заново всей работы по программированию. С другой стороны, если бы мы имели какие-нибудь средства *автоматической* трансляции таблицы решений вида, показанного на рис. 3.5, непосредственно в программу для ЭВМ (т. е. «препроцессоры» таблиц решений), то обо

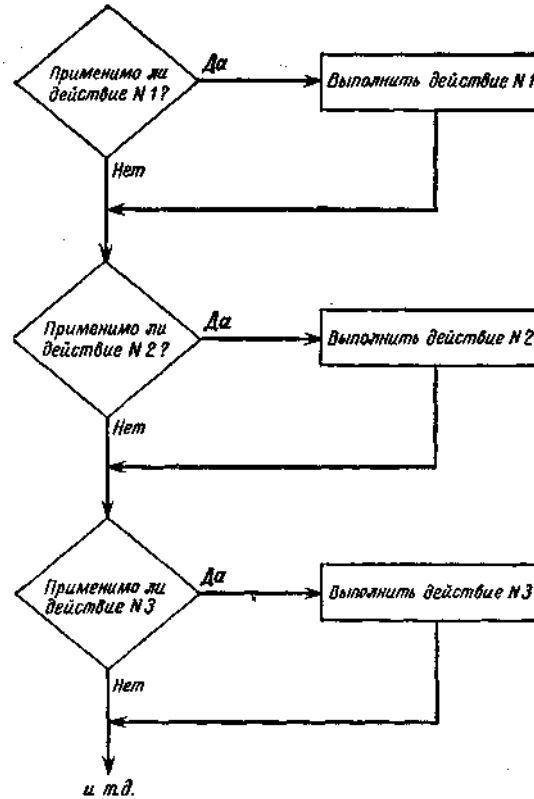


Рис. 3.6б. Общий случай преобразованной таблицы решений, представление в форме «применимо ли?».

всем этом можно было бы не беспокоиться. В этом случае мы, конечно, *предпочли* бы способ, иллюстрируемый рис. 3.5, так как небольшие изменения в требованиях пользователя привели бы, вероятно, к меньшим изменениям в правилах этой таблицы, чем в исчерпывающей таблице решений, показанной на рис. 3.2.

Подчеркнем еще раз, что в поставленной выше простейшей задаче не было нужды пользоваться формальным аппаратом таблиц решений. Однако в более сложных случаях задач этот подход может оказаться весьма полезным. Естественно, что нам хотелось бы добавить к нему многие усовершенствования: нам хотелось бы иметь возможность рассматривать ситуации, в которых условия могут принимать не только два значения; нам могли бы потребоваться таблицы решений, в которых условия комбинируются с помощью логической операции ИЛИ вместо логической операции И (или в дополнение к ней); нам хотелось бы исследовать различные

методы оптимизации записи таблиц решений, с тем чтобы программа выбирала соответствующие правила с использованием минимальных затрат времени ЦП и (или) памяти; мы могли бы обратиться к построению иерархий таблиц решений.

3.3.3. Используйте символьные параметры

Как было сказано выше, многие программисты действуют исходя из допущения (или даже данного им обещания), что им никогда не придется изменять их программу. В результате многие параметры программы кодируются в виде *констант;* таким образом, при изменении требований программист вынужден изменять каждый оператор исходной программы, в котором употреблена такая устаревшая константа.

Чтобы избежать этих трудностей, программист должен стремиться задать как можно больше параметров в символьной форме. Почти все языки программирования предусматривают возможность присваивать значения символическим именам, или константам, или некоторым «допустимым» выражениям. Фактическое присвоение значений обычно происходит во время ассемблирования или компиляции, хотя в иных случаях это может происходить в процессе редактирования связей (довольно неопределенный термин для обозначения действий, включающих обычно задание значений глобальных переменных в каждой из нескольких порознь скомпилированных программах) или во время загрузки программы в память, или даже в процессе исполнения программы. Хотя это может несколько замедлить процессы компиляции, трансляции с автокода или загрузки, использование символьных параметров обычно не замедляет процесса исполнения готовой программы и, *безусловно,* позволяет изменять данный параметр программы изменением только *одного* оператора в исходной программе.

Эта форма модульности может показаться довольно тривиальной, но ее не следует недооценивать. В большой программе на языке ассемблера значения этих параметров могут входить в сотни операторов; не пользуясь этим приемом, было бы довольно сложно изменить любую характеристику программы, а разве не с этим связано принятое нами определение модульности? Более важно то, что этот подход исключает случаи, когда программист забывает внести изменения в некоторые части программы, зависящие от значения данного параметра. Простой заменой одного оператора в исходной программе и повторной компиляцией (и, возможно, повторной загрузкой с другими порознь компилируемыми программами) мы автоматически обеспечиваем изменение всех затронутых этой заменой операторов исходной программы. Не пользуясь этим подходом, программист может случайяо пропустить какой-нибудь неясный оператор, в который следовало бы внести изме-

```
C
C     ЭТА ПОДПРОГРАММА БУДЕТ ВЫПОЛНЯТЬ ПОИСК ОПРЕДЕ-
C     ЛЕННОГО ЭЛЕМЕНТА В ПРОИЗВОЛЬНОМ ОДНОМЕРНОМ МАС-
C     СИВЕ. АРГУМЕНТЫ ЭТОЙ ПОДПРОГРАММЫ ПЕРЕЧИСЛЕНЫ
C     НИЖЕ:
C          TABLE ИМЯ МАССИВА, В КОТОРОМ ОТЫСКИВАЕТСЯ
                ЭЛЕМЕНТ
C          FIRST НАЧАЛЬНЫЙ ЭЛЕМЕНТ МАССИВА
C          LAST  КОНЕЧНЫЙ ЭЛЕМЕНТ МАССИВА
C          ARG   ЭЛЕМЕНТ, КОТОРЫЙ ДОЛЖЕН БЫТЬ НАЙДЕН
C          FLAG  ПОКАЗЫВАЕТ, БЫЛ ЛИ ПОИСК УСПЕЩНЫМ
      SUBROUTINE SEARCH (TABLE, FIRST, LAST, ARG, FLAG)
      DIMENSION TABLE (FIRST: LAST)
      •
```

и т. д.

Рис. 3.7а, Параметрическое определение размера таблицы в Фортран-программе.

нение. Некоторые мои студенты заметили, что тот же эффект достигается использованием хорошего пакета программ редактирования текстов в системах разделения времени. Обычно это так (хотя и не всегда; см. следующий раздел и рис. 3.9), однако порой мы наблюдаем, как небрежное определение текста исходной программы,

```
/ ПОДПРОГРАММА ПОИСКА В ТАБЛИЦЕ ОПРЕДЕЛЕННОГО ЭЛЕМЕНТА
/
/ НАЧАЛО ТАБЛИЦЫ И ДЛИНА ТАБЛИЦЫ ОПРЕДЕЛЕНЫ СИМВО-
/ ЛИЧЕСКИ, ТАК ЧТО ОНИ МОГУТ БЫТЬ ЛЕГКО ИЗМЕНЕНЫ.
          BEGIN = 200
          LENGTH = 300
          *BEGIN                / УСТАНОВИТЬ ЗНАЧЕНИЕМ СЧЕТЧИ-
                                  КА АДРЕСА НАЧАЛО ТАБЛИЦЫ
TABLE,    0
          *BEGIN + LENGTH       J ПЕРЕДАТЬ В СЧЕТЧИК АДРЕСА
                                  КОНЕЦ ТАБЛИЦЫ
SEARCH,   CLA                   / ОЧИСТИТЬ СУММАТОР
          DCA TEMP              / ЗАДАТЬ НАЧАЛЬНОЕ ЗНАЧЕНИЕ
                                  СЧЕТЧИКА
```

и т. д.

Рис. 3.7б. Программа на языке ассемблера PDP-8 с параметрическим определением размеры таблицы.

```
C
C     ЭТОТ ФРАГМЕНТ ПРОГРАММЫ НА ФОРТРАНЕ ОТЫСКИВАЕТ
C     В ТАБЛИЦЕ ЗАДАННЫЙ ЭЛЕМЕНТ. ОДНАКО ОН НЕ ЯВЛЯЕТСЯ
C     ПОДПРОГРАММОЙ
C
      DIMENSION TABLE (100)
      DO 10 I=1,100
      IF (ARG .EQ. TABLE (I)) GO TO 20
10    CONTINUE
C
C     ВОЙТИ ЗДЕСЬ В СЛУЧАЕ НЕУДАЧНОГО ПОИСКА
C     GO TO 30
C
C     ВОЙТИ ЗДЕСЬ, ЕСЛИ ЭЛЕМЕНТ НАЙДЕН
C
20    PRINT 100, I, TABLE (I)
```

Рис. 3.8а. Программа на Фортране, в которой не используется параметрическое задание размера таблицы.

подлежащего изменению, приводит к некоторым нежелательным заменам; так, вводя замену во всех случаях слова CAT словом DOG, мы обнаруживаем, что CATALOG превращается в DOGALOG!

Ниже приводятся некоторые простые примеры использования операции присваивания для задания значений параметров.

Размер таблицы. Число элементов в таблице, списке, очереди, буфере или области хранения промежуточных результатов может быть очень просто определено в параметрической форме. На рис. 3.7а

```
/ ПОДПРОГРАММА ПОИСКА В ТАБЛИЦЕ ЗАДАННОГО ЭЛЕМЕНТА
/
/ ЗАМЕТЬТЕ, ОДНАКО, ЧТО НАЧАЛО ТАБЛИЦЫ И ЕЕ ДЛИНА ОПРЕ-
/ ДЕЛЕНЫ КАК КОНСТАНТЫ
*1
          *200          / ПОСЛАТЬ В СЧЕТЧИК АДРЕСА НАЧАЛО
                          ТАБЛИЦЫ
TABLE,     0
          *500          / ПЕРЕДАТЬ В СЧЕТЧИК АДРЕСА КОНЕЦ
                          ТАБЛИЦЫ
SEARCH, CLA             / ОЧИСТИТЬ СУММАТОР
          TAD COUNT     / ЗАДАТЬ НАЧАЛЬНОЕ ЗНАЧЕНИЕ СЧЕТЧИКА
          DCA TEMP
          •

          и т. д.
```

Рис. 3.8б. Программа на языке ассемблера PDP-8, в которой не используется параметрическое задание размера таблицы.

```
* ЭТОЙ ПРОГРАММЕ ТРЕБУЮТСЯ ТРИ ТАБЛИЦЫ; РАЗМЕР КАЖДОЙ
* ТАБЛИЦЫ ЕСТЬ ФУНКЦИЯ ПАРАМЕТРА „РАЗМЕР". ЧТОБЫ ИЗМЕ-
* НИТЬ РАЗМЕР ТАБЛИЦ ДОСТАТОЧНО ПРОСТО ПЕРЕОПРЕДЕЛИТЬ
*„РАЗМЕР"
*
РАЗМЕР    EQU    40
TABLE1    DS     CL (РАЗМЕР)
TABLE2    DS     CL(2*РАЗМЕР)
TABLE3    DS     CL (РАЗМЕР+5)
```

Рис. 3.9. Программа с несколькими таблицами: язык ассемблера Системы IBM/360.

и 3.7б показаны фрагменты программ, записанных на Фортране и языке ассемблера, в которых размер таблицы определяется символом; на рис. 3.8а и 3.8б приводятся те же самые программы, определяющие размеры таблиц в виде константы. Этот способ оказывается особенно полезным, когда имеется несколько таблиц, буферов или очередей и т. д., размеры которых взаимосвязаны; на рис. 3.9 приводится пример подобной ситуации, записанный на языке ассемблера.

Относительные адреса элементов таблицы. Часто бывает желательным определить относительные адреса элементов таблицы или области хранения промежуточных данных в символьной форме. Так, вместо того чтобы на протяжении всей программы ссылаться на третий элемент таблицы посредством записи TABLE+3, ему следовало бы приписать независимое символьное имя. Рисунки 3.10а — 3.10в иллюстрируют удобство введения параметра в такой ситуации.

Отметим, что в Коболе эта задача решается использованием вложенных уровней в РАЗДЕЛЕ ДАННЫХ. В ПЛ/1 тот же результат дает использование *структур;* в некоторых языках это свойство настолько усилено, что позволяет программистам (или вынуждает их) ссылаться на элемент данных по собственному имени без какой-либо связи с другими физически близко расположенными элементами данных. Однако в большинстве версий Фортрана и в языке ассемблера такого рода параметризация требует от программиста некоторых осознанных усилий.

Константы. Допустим, что мы пишем программу, в которой мы неоднократно пользуемся константой 3,14159; мы могли бы ввести эту константу в программу с помощью одного из трех способов, показанных на рис. 3.11а—3.11в. (В этом примере использован язык Burroughs B5500 Алгол, но мы могли бы использовать и многие другие языки.) Отметим, что в реализации рис. 3.11а программист прилежно закодировал вручную значение я в нескольких различных операторах. Как мы отметили выше, это приводит к боль-

В прикладной задаче программист пишет на языке ассемблера программу обработки сообщений, получаемых с терминалов. Принятое сообщение вносится в *очередь входных элементов;* каждый элемент этой очереди имеет следующую структуру:

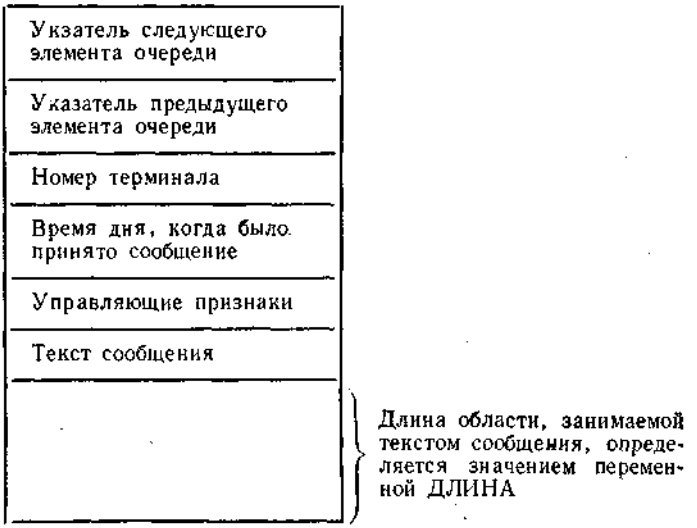


Рис. 3.10а. Параметрическое задание относительного расположения элементов таблицы.

```
NEXT     EQU   0        * УКАЗАТЕЛЬ СЛЕДУЮШЕГО ЭЛЕМЕНТА ОЧЕРЕДИ
PREV     EQU   1        * УКАЗАТЕЛЬ ПРЕДЫДУЩЕГО ЭЛЕМЕНТА ОЧЕРЕДИ
TERM     EQU   2        * НОМЕР ТЕРМИНАЛА
TIME     EQU   3        * ВРЕМЯ ДНЯ ПРИЕМА СООБЩЕНИЯ
FLAGS    EQU   4        * УПРАВЛЯЮЩИЕ ПРИЗНАКИ
TEXT     EQU   5        * НАЧАЛО ЗОНЫ ТЕКСТА
IENGTH   EQU   10       * ДЛИНА ЗОНЫ ТЕКСТА, ЧИСЛО СЛОВ

* В ЭТОМ РАЗДЕЛЕ ПРОГРАММЫ ПРЕДПОЛАГАЕТСЯ, ЧТО АДРЕС
* РАССМАТРИВАЕМОГО ЭЛЕМЕНТА ОЧЕРЕДИ УЖЕ БЫЛ ПЕРЕДАН
* В ИНДЕКС РЕГИСТР № 1
BEGIN    LDS   NEXT, 1   * ВЫБРАТЬ АДРЕС СЛЕДУЮЩЕГО ЭЛЕМЕНТА ОЧЕРЕДИ
         STS   NXTPNT    * УПРЯТАТЬ ЕГО
         LDS   TERM, 1   * С КОТОРОГО ТЕРМИНАЛА ПОСЛАНО СООБЩЕНИЕ?
         •
         •
         •
         и т. д.
```

Рис. 3.10б. Параметрическое задание относительного расположения элементов в таблице. Для ЭВМ типа GE-435, ориентированной на работу со словами, программист мог бы написать следующую последовательность команд обработки элементов очереди, показанных на рис. 3.10а,

Рис. 3.10в. Параметрическое задание относительного расположения элементов в таблице. Для ЭВМ, ориентированной на работу с байтами, подобной машинам типа Системы IBM/360, программист мог бы написать следующую последовательность команд обработки элементов очереди:

```
NEXT     EQU      0    УКАЗАТЕЛЬ СЛЕДУЮЩЕГО
                       ЭЛЕМЕНТА ОЧЕРЕДИ
PREV     EQU      4    УКАЗАТЕЛЬ ПРЕДЫДУЩЕГО
                       ЭЛЕМЕНТА ОЧЕРЕДИ
TERM     EQU      8    НОМЕР ТЕРМИНАЛА
TIME     EQU     12    ВРЕМЯ ДНЯ ПОЛУЧЕНИЯ СО-
                       ОБЩЕНИЯ
FLAGS    EQU     16    УПРАВЛЯЮЩИЕ ПРИЗНАКИ
TEXT     EQU     20    НАЧАЛО ЗОНЫ ТЕКСТА
LENGTH   EQU     40    ДЛИНА ЗОНЫ ТЕКСТА В БАЙ-
                       ТАХ

* В ЭТОМ РАЗДЕЛЕ ПРОГРАММЫ ПРЕДПОЛАГАЕТСЯ, ЧТО АДРЕС
* РАССМАТРИВАЕМОГО ЭЛЕМЕНТА ОЧЕРЕДИ УЖЕ БЫЛ ПЕРЕДАН
* В ИНДЕКС РЕГИСТР № 1
*
BEGIN    L       R2,NEXT(R1)   ВЫБРАТЬАДРЕССЛЕДУЮЩЕ-
                               ГО ЭЛЕМЕНТА ОЧЕРЕДИ
         ST      R2,NXTPNT     УПРЯТАТЬ ЕГО
         L       R2,TERM(R1)   СКОТОРОГОТЕРМИНАЛАПО-
                               СЛАНО СООБЩЕНИЕ?
         *
         •
         t
         и т. д.
```

```
ПЛОЩАДЬ:  =3.14159xРАДИУС*2;
ОКРУЖНОСТЬ:  =2X3.14159XРАДИУС;
IF X < 3.14159/2 THEN GO TO ОШИБКА;
IF X > 3.14159 THEN Y: =FALSE;
```

Рис. 3.11а. Параметрическое задание констант: B5500 Алгол фирмы Burroughs.

```
PI:  =3.14159;
ПЛОЩАДЬ:  = PI x РАДИУС*2;
ОКРУЖНОСТЬ:  =2XP1XРАДИУС;
IF X < PI/2 THEN GO TO ОШИБКА;
IF X > PI THEN Y: =FALSE;
```

Рис. 3.11б. Параметрическое задание констант: B5500 Алгол фирмы Burroughs.

```
DEFINE PI =3.14159#;
ПЛОЩАДЬ: =P1xРАДИУС*2;
ОКРУЖНОСТЬ: =2XP1XРАДИУС;
IF X > PI/2 THEN Y: =FALSE.
```

Рис. 3.11в. Параметрическое задание констант: B5500 Алгол фирмы Burroughs.

шим неудобствам в тех случаях, когда требуется изменить значение, константы (например, когда программист обнаруживает, что число 3,1415926535 более точно аппроксимирует константу *пи);* это означает также,что по небрежности он может забыть внести изменение в один из операторов исходной программы.

Рис. 3.11б определенно представляет вариант реализации, отличающийся большей модульностью; значение *пи* может быть легко изменено путем замены только того оператора исходной программы, которым переменной PI присваивается значение 3,14159. Однако такому способу свойствен тот потенциальный недостаток, что он влияет на время *исполнения программы,* т. е., после того как программа скомпилирована и загружена, необходимо исполнение оператора

PI: = 3,14159;

прежде чем значение переменной PI может быть использовано в каком-либо другом операторе. Обычно это требует исполнения одной или двух машинных команд *при каждом исполнении* этого фрагмента программы, чего программист, возможно, хотел бы избежать (особенно, если в его программе имеется несколько таких операторов). В то же время компилятор, оптимизирующий программу, мог бы выполнить это вычисление лишь однажды во время компиляции, так как он способен установить, что переменной PI присваивается значение только один раз.

На рис. 3.11в показан еще один способ. Значение *пи* определяется как значение параметра, но соответствующее присваивание выполняется *во время компиляции,* в виду чего не требуется дополнительного времени ЦП при выполнении программы. Оператором DEFINE в ходе процесса компиляции исполняется подстановка *строки знаков* 3,14159 вместо PI всюду, где PI встречается в исходной программе. Точно так же можно поступать в языке ПЛ/1, а используя макрооператоры и во многих наиболее развитых языках ассемблера. Заметим, в частности, что этот способ задания значения параметра оказался бы неэкономным при описании не константы, а *выражения,* так как это привело бы к необходимости заново вычислять его значение (и формированию компилятором соответствующих машинных команд) при каждом появлении рассматриваемого выражения в исходной программе. В таком случае было бы целесообразнее воспользоваться приемом, иллюстрируемым рис. 3.11б.

Заметим также, что существует много других способов задания значения параметра типа показанного на рис. З.Пв. В Коболе, например, удобно определять константы в секции WORKING STORAGE РАЗДЕЛА ДАННЫХ с помощью фразы VALUE. То же самое достигается в ПЛ/1 использованием атрибута INITIAL. Аналогично применение оператора DATA в Фортране дает возможность программисту присвоить в процессе компиляции начальное значение переменной (или массиву). В языке ассемблера программист при задании параметра обычно может выбирать или способ «подстановки строк», или более прямой способ, вроде того, что иллюстрирует рис. 3.116. В последнем случае программист обычно задает значение символьного имени *выражения,* если все элементы, входящие в выражение, были определены ранее.

Мы должны подчеркнуть, что независимо от выбора языка программирования или используемых методов способ программирования, показанный на рис. 3.11в, применять нецелесообразно. Как мы убедились выше, фактически каждый язык программирования располагает по крайней мере одним способом реализации параметрического подхода; дело программиста выбрать тот из них, который ему наиболее удобен.

Централизованное определение и задание значений параметров. Многие из существующих программных систем состоят из *нескольких* программ, каждая из которых написана разными лицами или группами лиц. Обычно они компилируются или транслируются порознь, могут обрабатываться редактором связей и загружаться, а также использоваться как совместно, так и раздельно (например, хотя эти программы и являются частями одной и той же составной программы, в основе одной из них может быть положена ежедневная обработка, другой — еженедельная, следующей — ежемесячная и т. д.). Так как эти программы являются частями одной и той же программы, весьма вероятно, что все они используют одни и те же таблицы, списки, константы и т. д.

Каждый программист, стремясь добиться большей степени модульности своей программы, мог бы ввести независимое определение значений параметров. Это, однако, легко могло бы привести к немодульному программированию, поскольку многие из параметров (например, константы, длины таблиц и т. д.) являются общими для всех составляющих программ. Если бы, например, мы решили изменить длину какой-нибудь общей таблицы, то мы должны были бы изменить операторы, которые задают значение соответствующего параметра, во всех программах, использующих эту таблицу, и затем заново компилировать или транслировать все измененные программы. К тому же вновь возникает риск, что мы забудем внести соответствующие изменения в одну-две второстепенные программы, которые, как выяснится, используют эту таблицу.

Ясно, что было бы предпочтительнее воспользоваться *общим*

Рис. 3.12а. Введение *глобальных переменных.* В примере на рис. 3.10 мы могли бы иметь отдельную программу, назовем ее PARAM, которая просто включала бы определения параметров. Таким образом, она состояла бы в следующем:

```
ENTRY   NEXT     * УКАЗАТЕЛЬ СЛЕДУЮЩЕГО ЭЛЕМЕНТА ОЧЕРЕДИ
ENTRY   PREV     * УКАЗАТЕЛЬ ПРЕДЫДУЩЕГО ЭЛЕМЕНТА ОЧЕРЕДИ
ENTRY   TERM     * НОМЕР ТЕРМИНАЛА
ENTRY   TIME     * ВРЕМЯ ДНЯ ПРИЕМА СООБЩЕНИЯ
ENTRY   FLAGS    * УПРАВЛЯЮЩИЕ ПРИЗНАКИ
ENTRY   TEXT     * НАЧАЛО ЗОНЫ ТЕКСТА
ENTRY   LENGTH   *ДЛИНАЗОНЫТЕКСТА.ЧИСЛОСЛОВ
•
•

любые другие определения



NEXT    EQU   0    * УКАЗАТЕЛЬ СЛЕДУЮЩЕГО ЭЛЕМЕНТА ОЧЕРЕДИ
PREV    EQU   1    * УКАЗАТЕЛЬ ПРЕДЫДУЩЕГО ЭЛЕМЕНТА ОЧЕРЕДИ
TERM    EQU   2    * НОМЕР ТЕРМИНАЛА
TIME    EQU   3    * ВРЕМЯ ДНЯ ПРИЕМА СООБЩЕНИЯ
FLAGS   EQU   4    * УПРАВЛЯЮЩИЕ ПРИЗНАКИ
TEXT    EQU   5    * НАЧАЛО ЗОНЫ ТЕКСТА
LENGTH  EQU   10   *ДЛИНАЗОНЫТЕКСТА.ЧИСЛОСЛОВ

любые другие определения
```

Примечание: В каждой строке исходной программы вам следует употреблять только один параметр ENTRY, поясняя в комментарии смысл вводимого параметра. Не следует помещать их в одну строку, т. е. писать

```
ENTRY NEXT, PREV, TERM, TIME, FLAGS, TEXT, LENGTH
```

так как эта запись неудобна в чтении, а также для внесения изменений.

для всех программ заданием значений параметров. То есть, если четырнадцать разных программ должны ссылаться на третий элемент таблицы (как в примере разд. 3.3.3), следует иметь лишь *одну* операцию введения параметра, вместо того, чтобы определять его в каждой из 14 программ. Аналогично, если для всех программ из таблицы требуется информация одного и того же вида (список разрешенных входов, программные коды допустимых форматов изменений, налоговые ставки, используемые программами оформ-

Рис. 3.126. Введение *глобальных переменных.* Все программы, пользующиесяпараметрами, определенными программой рис. 3.12а, должны быть написаны следующим образом:

```
        EXTERNAL    NEXT        * УКАЗАТЕЛЬ СЛЕДУЮЩЕГО ЭЛЕМЕНТА ОЧЕРЕДИ
        EXTERNAL    PREV        * УКАЗАТЕЛЬ ПРЕДЫДУЩЕГО ЭЛЕМЕНТА ОЧЕРЕДИ
        EXTERNAL    TERM        * НОМЕР ТЕРМИНАЛА
        EXTERNAL    TIME        * ВРЕМЯ ДНЯ ПРИЕМА СООБЩЕНИЯ
        EXTERNAL    FLAGS       * УПРАВЛЯЮЩИЕ ПРИЗНАКИ
        EXTERNAL    TEXT        *НАЧАЛОЗОНЫТЕКСТА
        EXTERNAL    LENGTH      * ДЛИНА ЗОНЫ ТЕКСТА, ЧИСЛО СЛОВ

* ПРОГРАММА ОБРАБОТКИ СООБЩЕНИЙ ДЛЯ МАШИНЫ GE-435 МОГЛА БЫ БЫТЬ НАПИСАНА СЛЕДУЮЩИМ ОБРАЗОМ, ПРЕДПОЛАГАЯ, ЧТО
* В ПЕРВОМ РЕГИСТРЕ ХРАНИТСЯ АДРЕС ТЕКУЩЕГО ЭЛЕМЕНТА ОЧЕРЕДИ.

        LDS         NEXT,1      * ВЫБРАТЬ АДРЕС СЛЕДУЮЩЕГО ЭЛЕМЕНТА ОЧЕРЕДИ
        STS         NXTPNT      * УПРЯТАТЬ ЕГО
        LDS         TERM,1      *СКОТОРОГОТЕРМИНАЛА ПОСЛАНО СООБЩЕНИЕ?
        •

LIST2   LDS         TEXT + 3,1  * ВЫБРАТЬ ТРЕТЬЕ СЛОВО ТЕКСТА

LIST3   LDS         TEXT + LENGTH-1,1  * ВЫБРАТЬ ПОСЛЕДНЕЕ СЛОВО ТЕКСТА СООБЩЕНИЯ
```

Примечание. В некоторых вычислительных системах возможности использования глобальных переменных несколько ограничены. Например, в'некоторых языках ассемблера запрещено употребление глобальных переменных в арифметических выражениях типа LIST3, использованных выше, т. е. выражений, в которых складываются или вычитаются два или более глобальных символических адреса. В некоторых языках ассемблера могут встречаться ограничения даже в употреблении арифметических выражений типа LIST2, т. е. содержащих глобальную переменную в сумме с локализованной переменной или константой.

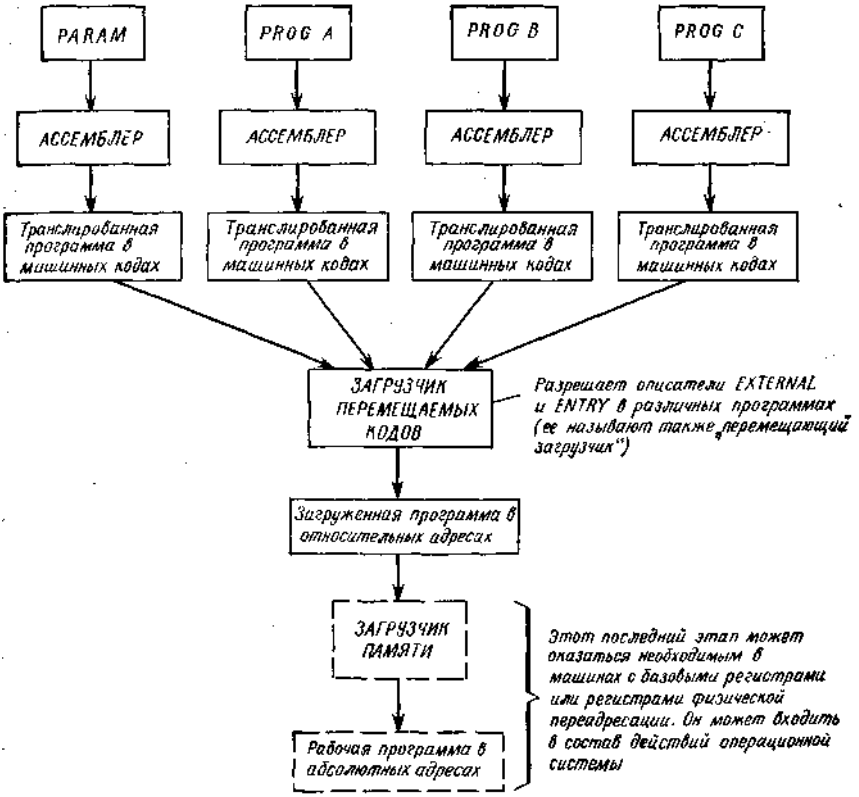


Рис. 3.12в. Процессы преобразования рассматриваемых программ,

ления счетов, коэффициенты в некоторых научно-исследовательских программах и т. п.), *все* эти программы должны обращаться к одной общей таблице.

Существуют два способа организации такого рода определения параметров. Простейший способ состоит в выделении операторов описания параметров в один файл исходной программы (например, отдельную колоду карт или специальный файл на магнитной ленте), который затем помещается в начале каждой из программ, которые должны компилироваться. Эта процедура обеспечивается применением оператора INCLUDE в большинстве версий КОБОЛа и аналогичными средствами, входящими в состав некоторых версий Фортрана, а также языка ПЛ/1.

Другой способ состоит в том, что все общие таблицы, списки, очереди и т. д. располагаются в *отдельной* программе, которая компилируется независимо от остальных. Все соответствующие символические имена параметров могут быть описаны затем как *гло-*

бальные переменные. При этом другие программы, использующие эти общие параметры, могут ссылаться на них как на *внешние глобальные* идентификаторы (применяемая здесь терминология довольно широко распространена, хотя в разных вычислительных системах могут наблюдаться некоторые специфические отличия). Внешние глобальные переменные «разрешаются»[1]), пользуясь еще одним сравнительно устоявшимся термином, в процессе редактирования или загрузки совместно компилируемых или транслируемых программ. Эта процедура иллюстрируется рис. 3.12а—3.12в.

3.3.4. Отделяйте действия по вводу-выводу от вычислительных операций

Многие программы для ЭВМ оказываются немодульными из-за того, что в них действия по вводу-выводу (I/O) рассеяны по всей программе. Поскольку весьма вероятно, что части программы, связанные с вводом-выводом, могут изменяться (так как по определению именно этими частями осуществляется интерфейс между программой и внешней средой), то очень важно, чтобы они были отделены от вычислительных частей программы. В большинстве случаев удобно писать программу таким образом, чтобы вычислительные модули могли вызывать данные *поэлементно;* в зависимости от конкретной задачи элементом данных может быть один символ в коде ASCII, образ карты, строка символов, число с плавающей точкой или одна запись базы данных. Подобным же образом программу следует писать так, чтобы подпрограммам выхода можно было пересылать по одному элементу выходных данных.

Ниже приводятся примеры некоторых действий, которые следует отделять от вычислительных операций программы.

Физические характеристики устройств ввода-вывода. Вычислительные разделы программы не должны зависеть от того, поступают ли входные данные с магнитной ленты, перфокарт или терминала; точно так же программа не должна зависеть от того, пересылаются ли выходные данные на быстродействующую печать, магнитную ленту или терминал. Это особенно важно потому, что выбор устройств ввода-вывода может измениться после того, как программа уже написана.

В большинстве современных операционных систем программист вынужден обращаться к *логическим,* образам устройств ввода-вывода, что препятствует использованию данных о физической природе этих устройств. И все же программист может легко попасть впросак, строя свою программу в расчете на то, что входные данные поступают с магнитной ленты, а не с перфокарт или терминала. В более конкретной ситуации программисты на Фортране часто пишут несколько операторов READ и WRITE, распределяя их

1) От слова «решать».— *Прим. перев.*

по всей программе и ссылаясь **в** каждом из них на определенный номер логического устройства ввода-вывода. Если устройство ввода-вывода изменяется (например, переходом от карт к ленте), то программисту приходится вносить изменения во все операторы READ и WRITE, которые ссылаются на логический номер этого устройства.

Правила инициирования ввода-вывода. Прежде чем прочесть первый элемент входных данных или записать первый элемент выходных данных, может оказаться необходимым открыть файлы,- сделать некоторые обращения к операционной системе, задать буферы, считать или вписать метки записей и т. п. Эти операции часто могут меняться (если, например, программы работают в условиях разных операционных систем), и поэтому их следует отделять от вычислительных модулей.

Выделение блоков и буферов. Как разбиение программы на блоки, так и выделение в ней буферов можно изменять с целью экономии памяти, уменьшения времени ЦП, занимаемого программой, лучшего использования физических свойств накопителей (например, устранения пропусков между записями на магнитной ленте) или адаптации к различным типам накопителей. Вместе с тем эти изменения не должны сказываться на разделах программы, обрабатывающих входные данные или формирующих выходные данные; их следует связывать только с работой подпрограмм ввода-вывода.

Способы исправления ошибок. На исправление ошибок ввода-вывода может потребоваться много времени. Ошибки *четности* в большинстве программ являются обычно наиболее частыми; возможны, однако, и многие другие типы ошибок. В некоторых случаях программист может полагать, что все ошибки будут устранены операционной системой. В иных случаях программист предпочтет исправление ошибок собственными средствами, еще в каких-то случаях он может попытаться устранить их *после* того, как операционная система выполнит соответствующие стандартные операции. Наконец, иногда программист *вынужден* сам заниматься устранением ошибок.

Как бы то ни было, устранение ошибок обычно является очень тонкой и сложной функцией многих вычислительных систем. Ее использование часто изменяется по мере того, как программист узнает больше о технических возможностях ЭВМ, об операционной системе и о требованиях по анализу ошибок в его задаче. Из-за многих возможных изменений крайне желательно, чтобы анализ и устранение ошибок были организованы в соответствии с принципом модульности. Для этого обычно необходимо различать *устранимые* и *неустранимые* ошибки. Примером устранимой ошибки может служить ошибка четности, вызванная пылинкой на магнитной ленте, в то время как на растянутой, смятой и исцарапанной в лентопротяжном устройстве ленте возможны неустранимые ошибки. Из соображений модульности рекомендуетсявсе действия с уст-

ранимыми ошибками выполнять в подпрограммах ввода-вывода, а все действия с неустранимыми — в вычислительных подпрограммах. Конечно, во многих случаях устранимые ошибки могут быть исправлены автоматически операционной системой (например, повторяя по несколько раз оператор READ), но для этого могут быть использованы и подпрограммы ввода-вывода.

Рис. 3.13а иллюстрирует эту точку зрения на примере короткого фрагмента программы на языке ассемблера мини-ЭВМ PDP-8; та же идея на примере Кобол-программы иллюстрируется рис. 3.136. Отметим, что в обоих случаях подпрограммы ввода-вывода возвращают управление при обнаружении неустранимой ошибки. В этой ситуации именноввычислительной программе должно быть принято решение о том, пропустить ли ошибочный входной (или выходной) элемент, остановить ли всю обработку, отпечатать ли специальное сообщение для оператора ЭВМ или предпринять какие-нибудь другие специальные действия. Если имеется несколько участков, на которых может потребоваться подобная обработка неустранимых ошибок (например, несколько точек, в которых вызываются одни и те же подпрограммы ввода-вывода и могут применяться одни и те же реакции на ошибки), то было бы естественным выполнять ее в общей подпрограмме.

```
IGNORE,  JMS READIN    / ВЫЗОВ ПОДПРОГРАММЫ ВВОДА
         JMP ERROR     / ВОЗВРАТ СЮДА В СЛУЧАЕ НЕУСТРАНИ-
                         МОЙ ОШИБКИ
         CLA           / ОЧИСТИТЬ СУММАТОР
         TAD BUFFER    / ВЫБРАТЬ ПЕРВОЕ СЛОВО ДАННЫХ
         '
         '
         '
/
/ВОЙТИ ЗДЕСЬ, ЕСЛИ ПРОИЗОШЛА НЕУСТРАНИМАЯ ОШИБКА. ЭТОТ
/ФРАГМЕНТ ПРОГРАММЫ ЗАСЫЛАЕТ ВОСЬМЕРИЧНОЕ ЧИСЛО 7777
/В СУММАТОР И ОСТАНАВЛИВАЕТСЯ. ЕСЛИ ОПЕРАТОР НАЖИМАЕТ
/КЛАВИШУ ПРОДОЛЖЕНИЯ РАБОТЫ, ТО ПРОГРАММА ПРОПУСТИТ
/ОШИБОЧНУЮ ЗАПИСЬ ВХОДНЫХ ДАННЫХ И ПЕРЕЙДЕТК ЧТЕНИЮ
/СЛЕДУЮЩЕГО ЭЛЕМЕНТА.
/
ERROR,   CLA CMA       / ПОСЛАТЬ 7777 В СУММАТОР
         HLT           / ЖДАТЬ, ВЫСВЕТИВ 7777
         JMP IGNORE    / ВЕРНУТЬСЯ И ПРОЧЕСТЬ СЛЕДУЮЩУЮ
                         ЗАПИСЬ
```

Рис. 3.13а. Программа обработки ошибок ввода-вывода для **PDP-8.**

```
NOTE    ПОЯСНИТЕЛЬНОЕ ЗАМЕЧАНИЕ. ПРЕДПОЛАГАЯ, ЧТО ВЫ-
        ЧИСЛИТЕЛЬНАЯ СИСТЕМА ПОЗВОЛЯЕТ ПРОГРАММИСТУ САМОМУ
        ОБРАБАТЫВАТЬ ОШИБКИ ВВОДА-ВЫВОДА, МЫ МОГЛИ БЫ НАПИСАТЬ
        ПРОГРАММУ, ПОКАЗАННУЮ НИЖЕ. МЫ ПРЕДПОЛАГАЕМ СУЩЕСТВОВАНИЕ
        ПОДПРОГРАММЫ. ИМЕНУЕМОЙ "READIN", КОТОРАЯ ВЫПОЛНЯЕТ ЧТЕНИЕ
        ВХОДНЫХ ДАННЫХ И УСТАНАВЛИВАЕТ ЗНАЧЕНИЕ ПРИЗНАКА "ERRFLAQ"
        РАВНЫМ 1, ЕСЛИ ПРИ ВВОДЕ ВСТРЕЧАЕТСЯ НЕУСТРАНИМАЯ ОШИБКА
GETNEXTREC.
        PERFORM READIN.
        IF ERRFLAG=1 THEN GO TO ERROR.
ERROR.
        DISPLAY ERRMESSAQE UPON CONSOLE.
        ACCEPT REPLY FROM CONSOLE.
        IF REPLY IS EQUAL TO YES THEN GO TO GETNEXTREC.
        STOP RUN.
```

Рис. 3.136. Обработка ошибок ввода-вывода в Коболе.

Изменение набора символов. Некоторая программа первоначально могла быть написана для обработки данных, записанных в коде ASCII; впоследствии могла возникнуть необходимость модифицировать ее для обработки данных, записанных в коде EBCDIC, BCD, Холлерита, или, быть может, символами специального подмножества некоторого данного алфавита (например, подмножества кодов ASCII по модулю 64). Если необходимы такие модификации, то мы предпочли бы избежать при этом изменения какой-либо из вычислительных подпрограмм, обрабатывающих входные символы и формирующих выходные символы.

Простейший путь достичь этого состоит в выборе одного стандартного внутреннего кода символов, который используют все вычислительные подпрограммы,— этим кодом может быть код ASCII, EBCDIC, специальный 6-битовый код(который был бы удобен в машинах с 12-, 18- и 36-битовыми словами) и т. п. При таком подходе следует написать модули ввода-вывода, переводящие входные данные во внутренний код, соответственно выходные данные должны быть переведены с внутреннего кода на любой код внешнего представления.

Заметим, что такой подход оказывается полезным в ряде других ситуаций, связанных с обработкой буквенной информации. Например, модулями ввода-вывода должна обеспечиваться стандартная реакция на символы, отмечающие конец сообщения, конец строки, а также другие управляющие символы, двойАые пробелы и т. д. **Большая** часть этих операций не относится к содержанию вычисли-

тельных модулей; так, в подпрограммах ввода-вывода может оказаться удобным удалить из входных данных управляющие символы, перед тем как передать их другим программам. Точно так же некоторые данные на входе могут иметь нестандартное представление (например, данные с телетайпа или перфоленты в зависимости от привычки пользователя могут заканчиваться символами «Возврат каретки — Перевод строки» или «Перевод строки — Возврат каретки»), однако в подпрограмме ввода-вывода должно быть обеспечено приведение этих данных к некоторой стандартной форме, прежде чем они будут переданы вычислительным модулям.

3.3.5. Не пользуйтесь общей рабочей памятью

В гл. 1 мы отмечали, что программисты часто пишут несколько модулей, использующих одни и те же ячейки памяти для хранения промежуточных результатов или значений переменных. Так, мы могли бы иметь подпрограммы A, B, C, D, использующие переменные TEMP1, TEMP2, по-видимому, для того, чтобы сэкономить память. Как мы видели в разд. 1.2.2, такая практика в программировании очень часто крайне затрудняет отладку.

Обычно значительно проще отлаживать программу, если каждому модулю отводится локализованная область для хранения промежуточных результатов. Это достигается естественным образом в таких языках, как Алгол и ПЛ/1, и в машинных языках, в которых легко реализуется память магазинного типа. Даже в простой мини-ЭВМ следует отказаться от выделения специальных ячеек памяти для хранения промежуточных результатов всех подпрограмм.

Мы должны подчеркнуть, что программа *может* быть написана с использованием общей рабочей памяти и *может* быть отлажена (хотя часто с определенными трудностями); *однако она не будет модульной*. Даже если подпрограммы могут использовать общую память промежуточного хранения в первоначальной версии программы, могут возникнуть неприятности при ее дальнейших изменениях и улучшениях. Модули, считавшиеся независимыми, вдруг оказываются в конфликте и начинают разрушать содержимое общей рабочей памяти. Заметим, что это чревато значительно большими трудностями в системах реального времени, коллективного пользования или системах многоцелевого назначения. Даже если модуль A не может непосредственно обращаться к модулю B, все же рискованно использовать в них одни и те же рабочие ячейки, так как модуль A может быть прерван модулем B, который затем начинает исполняться и разрушает содержимое ячеек, сформированное модулем A.

3.4. Стандартные подпрограммы

Одним из путей построения модульных программ является разработка стандартных библиотечных подпрограмм; эти подпрограммы могут затем применяться в качестве составляющих блоков индивидуальных прикладных программ специального назначения. Как и многие другие аспекты модульности, идея создания наборов стан* дартных подпрограмм весьма популярна, однако она редко используется на практике сколько-нибудь последовательно.

Тем не менее каждый программист, видимо, имеет интуитивное представление о понятии подпрограммы общего назначения; соответственно в этом разделе мы обсудим общность назначения с интуитивной точки зрения. Сначала мы обсудим некоторые преимущества и недостатки стандартных модулей; затем мы рассмотрим ряд ситуаций в программировании, в которых уместно воспользоваться методом стандартных подпрограмм.

3.4.1. Преимущества стандартных модулей

Стандартные модули экономят время программирования. В большинстве случаев средний программист скорее сможет найти стандартную версию программы, чем напишет ее сам. Так, если в некоторой организации снова и снова создаются модули одного и того же вида, то в ней следовало бы разработать собственную библиотеку стандартных модулей и естественно ожидать, что программисты должны быть знакомы с ее содержанием. Точно так же, если вы как индивидуальный программист снова и снова занимаетесь созданием подпрограмм одного и того же типа, то вам следует попытаться написать стандартную версию подпрограммы. Это позволит вам обзавестись набором «маленьких хитростей», которые упростят и ускорят ваши последующие программные разработки.

Хотя сознательно большинство программистов принимают эти доводы, подсознательно они их отвергают. По той ли причине, что стандартный модуль был написан, вероятно, кем-то еще или, как часто бывает, входит в состав поставляемого математического обеспечения ЭВМ, программист обычно не желает им пользоваться. Либо он не верит в этот модуль, либо слишком ленив, чтобы выяснить, как он работает. В психологическом отношении значительно интереснее и куда более привлекательнее создать собственную версию программы. Написание программы очень напоминает составление рисунка из хаотической кучи фрагментов, на которые он предварительно разрезан; использовать чью-то подпрограмму общего назначения равноценно тому, что кто-то решает для нас часть головоломки, а это в некоторой степени лишает вас удовольствия и чувства удовлетворения.

К тому же проявляется такое чувство, что «написание этой новой подпрограммы займет всего несколько минут, но потребуются *часы* на то, чтобы найти нужный стандартный модуль и понять, как он работает». Вздор и чепуха! Это распространенное заблуждение является одним из многих злосчастных проявлений переоценки собственных способностей в программировании. Написание этой подпрограммы займет *не* несколько минут: на ее проектирование потребуется час или около того, несколько часов займет кодирование и непредсказуемое количество времени программиста и ЦП — для ее отладки. Между тем на все это уходит несколько *дней* вполне *реального* времени!

Вы думаете, что бывает не так? Взгляните на последнюю программу, которую вы написали,— была ли она действительно столь уникальна, что вы не смогли бы воспользоваться некоторыми из подпрограмм поставляемого математического обеспечения? *Действительно* ли было невозможно использовать программы, написанные кем-либо другим в вашей организации? И наконец, сколько в *действительности* вам потребовалось времени, чтобы написать эту уникальную, вами лично созданную программу, которой вы так гордитесь?

Стандартные модули экономят память. Если каждый программист пишет свою собственную подпрограмму специального назначения, то вероятно появление большого числа сходных подпрограмм. Ясно, что если все эти программы используются в ЭВМ в одно и то же время, то это приводит к напрасному расходу больших объемов памяти.

Один из способов избежать этого состоит в применении нисходящей схемы программирования, рассмотренной в предыдущей главе. В начале проектирования программы не следует сосредоточивать внимания на отдельных подпрограммах; вместо этого основные функциональные элементы программы рассматриваются просто как «черные ящики». В процессе дальнейшего проектирования каждый из этих черных ящиков может быть расчленен на составляющие его компоненты; этот процесс продолжается до тех пор, пока не будет достигнут уровень отдельных модулей. В этот момент, должно быть, довольно просто определить, которые из модулей будут выполнять сходные функции. Должно быть также видно, какие из этих функций могут быть реализованы применением существующих, написанных ранее, стандартных подпрограмм.

В альтернативном подходе восходящего проектирования начинают на базе *существующей* библиотеки стандартных модулей. Обычно из них составляется нижний уровень модулей системы. Затем программист строит такие сочетания этих модулей, которые удовлетворяют требованиям его прикладной задачи. Такой подход не может быть успешным, если отсутствуют стандартные модули, которые могут быть адаптированы к требованиям рассматриваемой

задачи. При этом существует такая опасность, что характер задачи может насильственно «подгоняться» к требованиям, характеризующим существующие стандартные модули,— в таком случае уместно спросить, а являются ли эти модули на самом деле стандартными?

Использование стандартных подпрограмм защищает от ошибок программирования. Наконец, мы должны указать на то, что многие катастрофы в программировании могут быть исключены использованием стандартных модулей общего назначения. Специальные модули, написанные «на скорую руку», первое время могут казаться работающими правильно, однако в них может быть скрытая ошибка, следствия которой будут незаметны в течение нескольких дней, недель или месяцев. Это особенно чревато трудностями в тех случаях, когда программист пишет маленькую программу, использует ее раз или два и затем выбрасывает. Рассмотрим для примера простую программу, выполняющую копирование записей на магнитной ленте. Она могла бы быть написана программистом, которому нужно снять копию только с одной ленты, и, стараясь поскорее сделать эту работу, он мог пренебречь мерами по контролю ошибок. В результате, если во время чтения встречается ошибка четности, эта запись игнорируется и программист даже не печатает сообщения об ошибке. («В конце концов,— скажет программист,— я пишу всего лишь программу переписи этой единственной ленты и я знаю, что она хорошего качества, так что ошибка четности маловероятна».) И, конечно, к тому времени, когда кто-то заметит в новой копии что-то неладное, оказывается, что написанная «на скорую руку» программа потеряна или выброшена. В результате никто точно не знает, в чем именно была допущена ошибка, и исправить положение, видимо, будет невозможно.

В стандартных программах таких ошибок скорее всего не будет, так как программист вынужден проделать всю работу должным образом. И, если всеттаки что-то случается, можно было бы найти листинг программы и другие материалы и выяснить причину ошибки. Таким образом, психологически нас успокаивает уверенность в том, что, пользуясь стандартными программами, мы имеем дело с известным предметом; это должно приводить к предсказуемым результатам.

3.4.2. Потенциальные недостатки стандартных программ

Обсуждение стандартных программ будет лишено реалистичности, если не упомянуть об их недостатках. Мы отметили в предыдущем разделе, что отказ от метода стандартных программ обычно оправдывают леностью программиста и (или) его нежеланием использовать чьи-либо программы; имеются, однако, другие веские доводы, которые мы намерены обсудить ниже.

Стандартные программы бывает трудно найти. Иногда требуется *очень* много времени и терпения, чтобы отыскать в библиотеке подходящую стандартную подпрограмму. В большей мере это справедливо в тех случаях, когда программист обращается к специальным библиотекам ассоциаций пользователей (например, SHARE, DECUS и GUIDE), так как нет гарантий, что нужная программа работает, что она обеспечена сопровождением или что документация к ней отвечает требованиям. Трудности могут возникнуть и при попытках использовать стандартные модули, входящие в поставляемое математическое обеспечение ЭВМ.

Одна из основных проблем здесь связана с *документацией*. Документация, в которой дается описание того, как программа работает и как ею пользоваться, часто оказывается неполной, ошибочной или вообще отсутствует. В случае библиотеки, поставляемой с ЭВМ, которая может включать сотни и даже тысячи стандартных программ, трудность может заключаться в несоответствии ссылок, описаний и указателей. Хотя часто это оказывается за пределами возможностей отдельного программиста, мы все же должны отметить, что стоит потратить некоторые дополнительные усилия на организацию хорошо оформленной, простой в обращении библиотеки программ; эта дополнительная работа окупится сторицей в будущих программных разработках.

Стандартная программа может не удовлетворять принятым правилам программирования. Это может представлять особую трудность, если вы пытаетесь воспользоваться стандартной программой или процедурой, входящей в библиотеку ЭВМ или некоторой ассоциации пользователей. Правила программирования, применяемые автором стандартного модуля, могут отличаться от принятых в вашей организации. Могут по-разному пересылаться аргументы в подпрограмму, по-иному использоваться регистры и накопители; форма и содержание документации могут быть неприемлемыми, хотя и правильными.

В некоторых случаях стандартный модуль все же может быть использован после внесения изменений в порядок обращения, распределение регистров и т. п. Важно получить в результате полезную стандартную подпрограмму, которой можно пользоваться и вам, и другим программистам вашей организации. Таким образом, если даже вам придется полностью переписать исходную готовую подпрограмму, все же эти усилия и время могут оказаться потраченными не напрасно, если вы и ваши коллеги сможете воспользоваться ею еще раз.

Стандартные программы могут требовать значительно больше времени ЦП и памяти. К сожалению, это главный недостаток многих стандартных программ. Они просто слишком велики и медлительны для того, чтобы программисту хотелось бы ими воспользоваться. Известный пример «пакета», который попадает в эту категорию,

представляет собой пакет программ управления данными, поставляемый многими разработчиками ЭВМ и программного обеспечения,— одна из его версий занимает ни много ни мало 300K байт оперативной памяти!

Тем не менее вам не следует слишком торопиться выбросить стандартные программы, поставляемые с ЭВМ или же ассоциацией пользователей. В области вычислительных математических подпрограмм (например, SIN, COS, LOG, обращения матриц, преобразования Фурье) и в несколько меньшей степени в области прикладных процедур ввода-вывода стандартные программы можно было бы *оптимизировать* в смысле ускорения их работы и (или) занимаемой памяти. Написать второпях подпрограмму специального назначения, сравнимую по своим характеристикам со стандартной, может оказаться очень трудным делом. Если ваша специальная программа работает быстрее, то проверьте, выполняются ли в ней все действия по анализу ошибок, которые обычно предусматриваются в стандартных программах; т. е. проверьте, так ли хороша ваша работа, как эта стандартная программа. Кроме того, имейте в виду, что если стандартная программа написана хорошо, то она окажется и модульной: из нее можно в принципе удалить те части, которые определяют нежелательные свойства, повысив таким образом эффективность программы.

Часто бывает полезным опробовать стандартную программу и посмотреть, как она работает. Если оказывается, что она слишком медлительна или расточительно расходует оперативную память, диски или время загрузки каналов ввода-вывода, то вам, видимо, нужно написать свою собственную версию программы. Однако если вы *действительно* пишете свою собственную программу, то пишите ее так, чтобы ею могли пользоваться другие. Иными словами, сделайте ее стандартной программой!

3.4.3. Примеры программ, которые следует писать как стандартные

Мы приводим здесь лишь краткий список подпрограмм, программ и пакетов, которые должны быть оформлены как стандартные. Этот список не претендует на полноту, но он дает некоторое представление об общем подходе и точке зрения, которыми следует руководствоваться программисту, когда он пишет свою программу.

Подпрограммы поиска и просмотра таблиц. Типичной операцией в программировании для ЭВМ является *поиск* определенного элемента в таблице или, быть может, некотором файле на диске или барабане; необходимо также уметь *вносить* элементы в таблицу (или файл) таким образом, чтобы их можно было, снова легко найти. В большинстве случаев обе эти функции могут быть реализованы одной **и** той же программой. Для того чтобы внести новый элемент,

сначала мы должны попытаться его найти (возможно, безуспешно) и позаботиться о том, чтобы наша подпрограмма указывала адрес того места, где должен быть размещен новый элемент.

К сожалению, часто кажется, что каждый программист желает написать свою собственную подпрограмму поиска для каждой новой прикладной задачи, над которой он работает. Можно было бы легко написать стандартную подпрограмму поиска для использования всеми программистами данной организации. Ее можно было бы написать на Фортране, Коболе или ПЛ/1, с тем чтобы удовлетворить стандартным правилам, или на языке ассемблера для большей эффективности.

Подпрограммы ввода-вывода для мини-ЭВМ. В языках программирования высокого уровня программисты располагают мощными, удобными операторами для выполнения операций ввода-вывода. В больших и средних машинах даже в языке ассемблера программисту предоставляются удобные средства формирования соответствующих процедур: во всякой ЭВМ с операционной системой программисту-математику обычно не разрешаются прямые операции ввода-вывода, т. е. ему запрещено употребление машинных команд, непосредственно управляющих работой внешних устройств.

В мини-ЭВМ, однако, часто наблюдается совершенно иная ситуация. Хотя большинство из этих машин также оснащено операционными системами, их использование не является обязательным. Из-за специальных приложений (например, управление процессами или накопление данных) или ограничений по оперативной памяти, программист может отказаться от операционной системы и написать целиком свое собственное программное обеспечение. Поскольку программист имеет прямой доступ ко всем устройствам ввода-вывода, он часто доходит до написания своих собственных подпрограмм ввода-вывода. Это обычно очень расточительно, так как разработчик ЭВМ и ассоциация пользователей всегда поставляют все стандартные подпрограммы ввода-вывода для консольного телетайпа, устройств чтения перфолент, перфоратора, диска и т. п.

Сортировка и объединение. Стало привычным, что изготовитель ЭВМ поставляет как часть стандартного математического обеспечения пакет программ SORT (сортировка) и MERGE (объединение). Во многих случаях к этим пакетам возможно прямое обращение из Кобол-, Фортран-, Алгол- и ПЛ/1-программ. Однако эти готовые пакеты обычно размещаются на ленте или диске и представляются наиболее удобными для работы с большими массивами. Для упорядочения небольших совокупностей элементов, например таблицы, содержащей несколькосотен илименьшее числоэлементов, программист мог бы пожелать составить свою собственную подпрограмму сортировки. Мы должны, однако, предупредить еще раз, что если в организации имеется десять программистов, которым требуется небольшая программа сортировки, то каждый из них будет писать

свою собственную; хуже того — каждый раз, когда некоторому программисту потребуется подпрограмма сортировки, он напишет ее заново. Очевидно, что следовало бы написать стандартную подпрограмму сортировки.

Дампы, трассировка, «мгновенные фотографии», пакеты отладочных программ и т. п. Среди программистов, работающих на языке высокого уровня, таких, как Кобол и Фортран, наблюдается тенденция использовать встроенные отладочные операторы; примерами могут служить операторы DUMP и PDUMP в Фортране и оператор MONITOR в B5500 Алголе фирмы Burroughs. В языке ассемблера, однако, программист часто вынужден сам писать программы трассировки, «мгновенной фотографии» и т. п. Помимо простых средств трассировки и дампинга, программисту могут потребоваться довольно содержательные пакеты тестовых программ: программы генерации тестовых входных данных; программы генерации тестовых файлов и баз данных; отладочные программы ДДТ-типа и т. п. Если многие программисты одной и той же организации отлаживают программы одного и того же типа (или, что эквивалентно, если программист рассчитывает снова и снова описать и отлаживать программы одного и того же типа), то может иметь смысл создание набора стандартных тестовых программ. В гл. 7 и 8 обсуждается ряд полезных тестовых и отладочных пакетов программ.

Подпрограммы перевода. Во многих прикладных задачах программисту приходится выполнять различного рода *переходы:* от кода ASCII к коду BCD; от кода ASCII к некоторому внутреннему набору символов, такому, как набор 6-битовых символов в машинах фирм Honeywell и Burroughs; или от кода ASCII к двоичному, от десятичных чисел к восьмеричным и т. п. Изготовители ЭВМ часто поставляют большое число таких подпрограмм, написанных в достаточно стандартной форме; многие подпрограммы перевода могут быть также найдены в библиотеках ассоциаций пользователей.

Простые вспомогательные подпрограммы ввода-вывода. Всякая вычислительная система должна быть снабжена программами обмена лента — АЦПУ, карты — диск и диск — лента; должны быть также программы копирования записей на лентах, реперфорации карт и т. п. Эти программы тоже обычно поставляются изготовителем ЭВМ; однако из-за специальных правил разметки, требований к формированию блоков и буферов или особенностей задания форматов в каждой конкретной системе может потребоваться свой собственный набор вспомогательных программ. Важно, конечно, при этом избегать таких ситуаций, когда каждый программист пишет свои собственные вспомогательные программы; или еще худшей ситуации, когда каждый программист пишет каждый раз новый набор вспомогательных программ, аналогичный» предыдущему или слегка от него отличающийся.

Подпрограммы динамического распределения памяти. В больших вычислительных системах, в которых оперативная память и память на дисках или барабанах должна распределяться динамически, следует иметь одну стандартную подпрограмму, которой могут пользоваться все программисты; следует иметь также подпрограмму освобождения памяти, когда она больше не требуется.

ЛИТЕРАТУРА

1. Harding D. A., Modular Programming-Why Not?, The Australian Computer Journal, v. 4, №4, Nov. 1972, p. 150—156.
2. Blee M., Modular Programming-Innovation or Common Sense?, Data Systems, Feb. 1969, p. 26—27.
3. Cohen A., Modular Programs: Defining the Module, Datamation, Jan. 1972, p. 34—37.
4. Judd R., Practical Modular Programming, Computer Bulletin, v. 14, p. 4—7.
5. Klingels I., Some Thoughts on the Future of Modular Programming, Data Processing, v. 13, July-Aug. 1971, p. 268—269.
6. Packer D. W., Effective Program Design, Computers and Automation, July 1970, p. 37—41.
7. Rhodes J. J., Management by Module, Data Systems, Aug. 1971, p. 34—36.
8. Rhodes J. J., Management by Module, Data Systems, Sept. 1971, p. 36—38.
9. Rhodes J.J., Management by Module, Data Systems, Oct. 1971, p. 32—34.
10. Armstrong R., Modular Programming in COBOL, Wiley, Inc., 1973.
11. Madnick S., Alsop J., Modular Approach to File System Design, Proceedings of 1969 SJCC, p. 1—13.
12. Maynard J., Modular Programming, Auerbach Publ., 1972.
13. Myers G. J., Characteristics of Composite Design, Datamation, Sept. 1973, p. 100—102.
14. Parnas D. L., A Technique for Software Module Specification with Examples, Communications of the ACM, May 1972, p. 330—336.
15. Parnas D. L., On the Criteria to Be Used in Decomposing Systems into Modules, Communications of the ACM, Dec. 1972, p. 1053—1058.

ВОПРОСЫ

1. Дайте краткое определение модульного программирования.
2. Дайте краткое определение модуля.
3. Если бы вам пришлось изучать чью-то программу, то каким бы образом вы определили, что она модульна? Если бы вам пришлось изучать две программы, А и Б, то каким бы образом вы определили, что программа А более модульна, чем программа Б?
4. Какая преследуется цель при составлении модульных программ? Считаете ли вы, что эта цель очевидна для пользователя типичной вычислительной программы или системы?
5. В разд. 3.1.1 предполагается, что модульность программы может быть выражена через понятие *размера* модуля. В чем преимущество такого определения? В чем его недостатки? Считате ли вы, что такое определение модульной программы чревато неверным его употреблением, и если да, то каким образом?
6. Считаете ли вы, что размер модуля необходимо ограничивать двадцатью программными операторами? Аргументируйте ваше мнение.
7. Существенно ли ограничение размера модуля пятьюдесятью операторами? Находите ли вы это ограничение разумным? Согласились ли бы вы соблюдать его?
8. Считаете ли вы разумным ограничение размера модуля объемом месячных

затрат труда среднего программиста? Не следует ли, на ваш взгляд, ограничить его меньшим объемом трудозатрат, например двухнедельным?

9. Что мы имеем в виду говоря, что один модуль «не_зависит» от другого модуля? Можно ли это выразить количественно? Можете ли вы по виду программы сказать, являются ли ее модули независимыми? Хотелось бы вам научиться это делать?

10. Каким образом вы можете определить, что программа зависит от некоторого частного алгоритма? Приведите пример программы, в которой изменение некоторых составляющих общего алгоритма повлекло бы необходимость измене-- ния нескольких модулей.

11. Каким образом вы можете определить, что программа сильно зависит от определенного набора аргументов или параметров? Приведите пример программы или модуля, в котором некоторое изменение аргументов (например, переход от формата обычной точности к формату двойной точности) вызывало бы большие изменения в модуле.

12. Каким образом вы можете определить, что программа сильно зависит от внутренних таблиц, констант или переменных? Считаете ли вы, что это — часто встречающаяся ситуация? Приведите пример, когда изменение формата определенной таблицы или константы влекло бы за собой необходимость значительных изменений в программе.

13. Каким образом вы можете определить, что программа сильно зависит от структуры базы данных? Приведите пример такой программы в вашей организации. Что может быть сделано для уменьшения зависимости программы от форматов и структуры базы данных?

14. Каким образом вы можете установить, что программа сильно зависит от таких управляющих характеристик, как реентерабельность, рекурсивность или повторное использование модулей? Дайте примеры соответствующих программ в вашей организации. Что может быть сделано для уменьшения зависимости программы от факторов этого сорта?

15. Кратко разберите преимущества модульного программирования. Считаете ли вы, что эти преимущества осознают и оценивают в вашей организации?

16. В чем недостатки модульного программирования? Значительны ли проявления этих недостатков в вашей организации? Что может быть сделано для их минимизации?

17. Считаете ли вы, что большинство программистов в вашей организации понимает смысл модульности? В качестве эксперимента проведите опрос среди программистов, с которыми вы работаете. Спросите их, как они себе представляют модульную программу? Совпадают ли их ответы?

18. Попытайтесь найти пример программы, написанной кем-нибудь в вашей организации, которая сначала считалась (автором программы) модульной, но впоследствии таковой не оказалась. Какими характеристиками программы объясняется ее недостаточная модульность? Осознал ли автор программы свои действия, т. е. *почему* ему казалось, что он пишет модульную программу?

19. Намного ли больше времени требует разработка модульных программ? В качестве эксперимента возьмите относительно несложную задачу на программирование; найдите приятеля или вашего коллегу приблизительно равной с вашей квалификацией в программировании; напишите сами модульное решение задачи и попросите вашего приятеля написать его версию программы как можно быстрее. Насколько больше (если вообще больше) времени потребовалось на разработку вашей программы? На ваш взгляд, стоит ли это дополнительных усилий и затраченного времени?

20. В большинстве случаев практика модульного программирования предполагает использование некоторых средств вызова подпрограмм, например, оператора PERFORM в Коболе, оператора CALL в Фортране или команды BALR в языке ассемблера в Системе IBM/370. Изучите средства вызова подпрограмм в используемом вами языке программирования. Сколько микросекунд требуется на их исполнение? Считаете ли вы, что это много?

21. Повторите эксперимент, предложенный в п. 19. На сколько больше времени ЦП требует исполнение вашего модульного образца программыв сравнении с версией, составленной «на скорую руку»? Считаете ли вы, что это много? Если модульный вариант программы *действительно* требует больше времени ЦП на исполнение, считаете ли вы это оправданным другими его достоинствами?

22. Что может быть сдаадно для минимизации дополнительных затрат времени ЦП, связанных с модульным программированием? На каком этапе работы это следует делать?

23. Рассмотрите еще раз п. 20; на этот раз, однако, определите объем оперативной памяти, занимаемой (в объектном коде) средствами вызова подпрограмм в используемом вами языке программирования. Считаете ли вы, что это много?

24. Повторитеэксперимент, предложенный в п. 19. Насколько больше памяти требует ваш модульный образец программы в сравнении с версией, составленной «на скорую руку»? Считаете ли вы, что это большие затраты? Если модульный вариант программы *действительно* занимает больше памяти, считаете ли вы это оправданным другими его достоинствами?

25. Используются ли таблицы решений в вашей организации? Если нет, то почему? На ваш взгляд, насколько трудно было бы убедить программистов пользоваться таблицами решений?

26. Почему таблицы решений ведут к более модульным программам? Необходимо ли предусмотреть какие-нибудь специальные шаги, чтобы убедиться, что окончательные программы действительно будут модульными?

27. Приведите пример программы, в котором недостаточное использование символических параметров (рассмотренное в разд. 3.3.3) обусловливает немодульность программы. Представляет ли какие-нибудь трудности введение в программу символических параметров?

28. В чем состоит достоинство описания размера таблицы с помощью символического параметра? Как следует поступать, если в программе содержится несколько таблиц, размеры которых взаимосвязаны (например, одна из таблиц всегда вдвое длиннее другой)?

29. В чем заключаются возможные осложнения при неиспользовании символических параметров для описания тех свойств программы, которые могут изменяться?

30. Рассмотрите два различных способа определения констант в программе как символических параметров. В каком способе требуется больше времени ЦП? Какой из них удобнее?

31. Почему действия по вводу-выводу следует отделять от вычислительных разделов программы? В чем преимущество этой точки зрения?

32. Как может быть спроектирована программа, чтобы ее логика не зависела от изменений в физических характеристиках устройств ввода-вывода? Встречаются ли ситуации, когда это трудно осуществить?

33. Какможетбытьспроектирована программа, чтобы ее логика не зависела от способа выделения буферов и блоков? Трудна ли эта задача в реализации?

34. Почему нам следует строить наши программы так, чтобы модули не использовали общих зон памяти временного хранения? Дайте описание вида ошибки программирования, которая может возникнуть в результате такого разделения памяти.

35. Некоторые языки программирования позволяют довольно легко избежать этих трудностей с общей памятью временного хранения. В Алголе, например, каждый модуль может определить свои собственные «локализованные» переменные, которые известны только в границах определившего их модуля. Таким образом, если модуль GLOP определяет локализованную переменную X, то она будет иметь иное значение (и будет связана с иным размещением в памяти), чем *локализованная* переменная X, определенная в модуле FLOP. Обладает ли такими свойствами ваш язык программирования? Есть ли в этом какие-нибудь недосгатки? Каким образом можно добиться того, чтобы несколько программистов, работающих над одним и тем же проектом, пользовались этим средством?

36. В некоторых языках отсутствует понятие «локализованной» переменной, упомянутой в п. 35. Все переменные и области памяти, описанные в программе, доступны всем входящим в нее модулям. Каким образом в подобной ситуации можно достичь того, чтобы каждому модулю была отведена отдельная область рабочей памяти? Каким образом мы можем добиться уверенности в том, что несколько программистов, работающих над одним и тем же проектом, предпринимают соответствующие меры по выделению для их модулей независимых зон памяти временного хранения?

37. Считаете ли вы, что использование стандартных модулей значительно сэкономит время программирования? Наблюдали ли вы, чтобы написание специальной версии программы оказалось для вас проще и быстрее?

38. Считаете ли вы, что использование стандартных программ может привести к экономии памяти? Считаете ли вы, что стандартные программы (например, программа вычисления корня квадратного, поставляемая изготовителем ЭВМ) будут исполняться быстрее, чем специальные версии программ, которые вы могли бы написать сами?

39. В чем состоят преимущества библиотеки стандартных программ? Оцениваются ли эти преимущества в вашей организации? Требуется ли от программистов использование стандартных программ во всех возможных случаях? Если нет, то почему?

40. В чем состоят потенциальные недостатки стандартных программ? Существенны ли эти недостатки? Могут ли они встретиться в практике среднего программиста?

ГЛАВА **4**

СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ

4.0. Введение

В течение нескольких лет многие ученые, работающие в области вычислительной науки, принимали участие в изучении и совершенствовании сравнительно нового подхода к проектированию и написанию программ — концепции, известной как *структурное программирование*. С этим подходом связывают надежды на существенный прогресс в программировании, который позволит уйти гораздо дальше существующих здесь специальных приемов, причем те немногие эксперименты, в которых структурное программирование было опробовано, оказались весьма успешными. И все же, несмотря на довольно широкое освещение его в мировой литературе и посвященные ему дискуссии, которые все еще продолжаются, этот подход остается неизвестным подавляющему большинству программистов, особенно в Соединенных Штатах. И если они *все-таки* что-то слышали о нем (обычно в крайне упрощенной форме), то склонны его отвергать как неоправданно ограничительный и нецелесообразный.

Таким образом, одна из основных задач этой главы состоит в описании основных предпосылок, из которых исходит движение сторонников структурного программирования, с тем чтобы его цели стали более понятными. Этими целями всегда были большая «читабельность» и ясность программ, более высокая производительность программистов и упрощение процесса испытаний. Затем мы сосредоточим наше внимание на методах и теории структурного программирования, причем будут приведены примеры приложений на общепринятых языках программирования.

4.1. Основные предпосылки структурного программирования

4.1.1. Ранние работы Дейкстры и других

Профессор Э. Дейкстра (Эйндховен) был одним из первых инициаторов структурного программирования. В 1965 г. на конгрессе IFIP [1] он высказал предположение, что оператор GO TO мог бы

быть исключен из языков программирования. Больше всего, вероятно, запомнилось на конгрессе его заявление, что квалификация программиста обратно пропорциональна числу операторов GO TO в его программах![1] Несмотря на то что конгресс IFIP 1965 г. происходил в Нью-Йорке и отличался многочисленностью участников из целого ряда стран, это заявление вызвало сравнительно слабую реакцию. Многие программисты только что отказались от пользования Фортраном II и связывали свои дальнейшие планы с переходом на Фортран IV, одним из краеугольных камней которого является почтенный оператор GO TO. Другие переживали болезненный рост, переходя от РПГ к Коболу, третьи были поглощены изучением только что появившейся Системы IBM/360.

Будучи не из тех, с кем можно не посчитаться, Дейкстра повторно изложил свои идеи в письме редактору *Communications of ACM* в марте 1968 г. [2]. Он сформулировал также некоторые из своих идей по нисходящему проектированию систем (которые, по-видимому, развиваются параллельно и в согласии с идеей структурного программирования) в статье, представленной Первому симпозиуму по основам операционных систем,— статье, перепечатанной в последствии в майском номере *Communications of ACM* за 1968 г. [3]. Тогда же он принял участие в организованных НАТО конференциях по методам программного обеспечения и изложил некоторые из своих идей в *Software Engineering Techniques,* 1969 [4]; именно в этом отчете впервые появилось его замечательное сравнение программы с жемчужным ожерельем. Другая заметка, содержащая описание его идей, была издана Эйндховенским университетом [5].

Цели, которые преследует Дейкстра во всей своей работе, представляются весьма постоянными. Как он указывает в своей статье в журнале *Software Engineering Techniques* [4].

Основным был вопрос: возможно ли повысить на порядок уровень наших способностей в программировании и какие средства (теоретические, организационные и технические) можно было бы использовать в процесс создания программы, чтобы достичь этого уровня. Больше, чем что-либо другое, Дейкстру, по-видимому, волновала проблема доказательства правильности программы для ЭВМ, т. е. разработка математически строгих методов доказательства правильности программ, которые бы устранили необходимость в дорогостоящих, кропотливых и чаще всего недостаточных специ-

1) О Дейкстре (так же как о Кнуте и других системных суперпрограммистах) ходит множество историй, и многие из его замечаний уже составляют фольклор вычислительной науки. Известно, что он мало заинтересован в приеме на старшие курсы университета, где он преподает, студентов со знанием Фортрана по той причине, что вместе с этим знанием могли привиться дурные привычки программирования. От него пошло также высказывание, что если знание Фортрана можно сравнить с младенческим расстройством, то уж ПЛ/1 — определенно роковая болезнь.

альных приемах их испытании. В той же статье он формулирует это так:

«...Я сосредоточил свое внимание не на вопросе «как мы доказываем правильность данной программы», а на вопросе «какими должны быть структуры программ, чтобы без чрезмерных усилий мы могли находить доказательство их правильности, если даже программы оказываются большими?», и как следствие этого на вопросе «как нам составить для данной задачи такую хорошо структурированную программу?». Мое стремление рассматривать лишь такие «хорошо структурированные» программы (как подмножество всех возможных программ) основывается на уверенности в том, что мы можем найти такое подмножество, которое удовлетворяет нашим нуждам программирования, т. е. что для всякой программируемой задачи в этом подмножестве найдутся в достаточном количестве реалистические программные решения».

В это же время определенный интерес к проблеме проявляли и другие исследователи, хотя их чаще интересовали вопросы, касающиеся языков программирования и теории формальной вычислимости, чем практические приложения структурного программирования. Профессор Вейнгартен заметил, что в Алголе 60 оператор GO TO можно было бы исключить использованием подходящего препроцессорного алгоритма [61. Несколько авторов ввели специализированные языки и стили программирования, которыми исключается оператор GO TO, среди них — разработка LISPX и MOL-32 Ван Шорра [7] и ISWIM Ландина [8].

4.1.2. Исследования IBM

На той же самой конференции по методам программного обеспечения, на которой Дейкстра сообщил некоторые из своих идей по структурному программированию, Дж. Арон, фирма IBM, докладывал об эксперименте, известном под названием «проект суперпрограммиста» [9]. В этом эксперименте одному программисту, д-ру Х. Миллсу, было предложено проделать нечто почти невозможное — выполнить за шесть месяцев работу, представляющую, как оказалось, проект на 30 человеко-лет. Хотя было сказано лишь то, что эксперимент был успешным (д-р Миллс, например, был скрыт от пользователей, чтобы их не смущал тот факт, что две группы программистов работали над одним и тем же проектом!), его результатом явились некоторые новые идеи в организации разработки, проектировании систем и в проектировании программ, прежде всего — идеи нисходящего проектирования, которые мы рассматривали в гл. 2. Кроме того, были выполнены работы по языку ПЛ1, который, как мы увидим ниже в этой главе, обладает удобными средствами реализации структурного программирования.

Успех этого эксперимента побудил IBM испробовать те же идеи в более крупной разработке—проектеинформационно-поисковой системы для газеты *Нью-Йорк Таймс* [10]. Так же как и в предыдущем случае, основные усилия в этом проекте были посвящены поиску новых эффективных методов организации и управления разработкой, хотя значительная роль отводилась также структурному программированию и нисходящему программированию. Этот проект был весьма интересен тем, что он отличался сравнительно большим масштабом (около 83 000 операторов исходной программы), практическим характером (реальный заказчик платил реальные деньги за работающую систему) и успешной реализацией (производительность программистов оказалась приблизительно в пять раз выше производительности среднего программиста).

После успешного завершения проекта для *Нью-Йорк Таймс* отделение федеральных систем фирмы IBM, в котором впервые зародилась идея использования «суперпрограммиста», начало внедрять свои идеи в других отделениях IBM и в организациях нескольких крупных пользователей продукции фирмы IBM. Многие крупные правительственные учреждения проявили большой интерес к этим идеям (правда, опять же, основное внимание, кажется, было уделено управленческим аспектам подхода фирмы IBM), так что есть некоторые основания надеяться, что вскоре они овладеют и основной массой «рядовых» пользователей систем IBM.

4.1.3. Последние результаты и существующие мнения

Помимо работ, упомянутых выше, во многих университетах и исследовательских организациях были начаты эксперименты по программированию без GO TO — одно из крайне упрощенных представлений о структурном программировании. В частности, сотрудниками университета им. Карнеги — Мелона был создан «язык для разработки систем», названный BLISS, в котором нет оператора GO TO. По мнению проф. У. Вулфа, опыт трехлетней работы на языке BLISS и его использование при создании компиляторов и операционных систем показали, что это — удобный и полезный язык программирования; по логике вещей надо полагать, что опыты структурного программирования также оказались успешными. В частности, Вулф сообщает [11], что «...Неизбежный вывод из практики использования BLISS'а состоит в том, что якобы существующее неудобство программирования без оператора *go to* является мифом. Программисты, знакомые с языками, содержащими оператор *go to* проходят довольно короткий и безболезненный период адаптации. После такойадаптации они обнаруживают, чтоотсутствиеоператора *go to* не является помехой; напротив, неизменно отмечается, что введенная дисциплина программирования без оператора *go to* структуризует и упрощает задачу».

Замечания Вулфа были высказаны на 25-й Национальной конференции ACM в 1972 г.; эти замечания составили часть материалов одного из заседаний, посвященных этому спорному вопросу программирования, которое было названо «Полемика по GO TO». После ряда статей и устных выступлений как за, так и против изъятия оператора GO TO, значение которого будет более ясным после разд. 4.3, стало очевидным, что мнения остались неизменными — те, кто прибыли на заседание сторонниками оператора GO TO, сторонниками его и покинули, а те, кто были против, противниками и остались. Говоря словами М. Гопкинса [12], «... наша мудрость еще не достигла той ступени, на которой в будущих языках можно было бы исключить оператор *go to*. Если будущие исследования покажут, что, избегая употребления *go to,* мы можем достичь некоторых важных преимуществ, то решение сохранить конструкцию *go to* может быть пересмотрено. Но до той поры ее разумнее сохранить.»

Тем не менее кажется, что большинство специалистов в области обработки данных постепенно обращаются к идеологии структурного программирования. Журнал *Datamation* в декабрьском номере 1973 г. назвал структурное программирование «революцией в программировании» и посвятил ему пять статей [33, 37]. Дейкстра, Дал и Хоор [39], а также Вейнберг [381 написали на эту тему книги. В нескольких университетах было введено обучение структурным методам программирования, что оказалось особенно успешным в отношении тех студентов, которые прежде не имели накакого опыта в программировании. Университеты, научно-исследовательские организации, разработчики ЭВМ (в частности, фирма IBM, хотя прочие фирмы участвуют в этом менее активно) и ученые в области вычислительной науки прилагают усилия в части накопления практического опыта структурного программирования. В то же время они пытаются найти способы применения этого подхода, позволяющие упростить задачу тестирования программ для ЭВМ. Между тем, однако, многие программисты видят в этой идее лишь один спорный аспект — отказ от оператора GO TO — и без дальнейших рассуждений отвергают ее. Можно ожидать, что будущие исследования и эксперименты послужат подтверждением более высокой эффективности и управляемости программ, написанных в духе этой идеологии; мы можем ожидать, что тогда она будет принята большей частью опытных программистов и системных аналитиков.

4.2. Назначение и истоки структурного программирования

Здесь читатель может указать на то, что он все еще так и не знает, в чем состоит структурное программирование; хужетого, он, может быть, уже пришел к выводу, что оно просто сводится к исключению из языков программирования оператора GO TO, и с возмущением

бросил чтение этой главы и перескочил дальше к гл. 5, чтооы узнать, нет ли в ней чего-нибудь более полезного!

На самом деле структурное программирование представляет собой нечто большее, чем один лишь отказ от оператора GO TO, и мы намерены обсудить это более подробно (в разд. 4.3). Тем не менее мой опыт изложения этого предмета различным группам безразличных (а иногда и враждебно настроенных!) к нему программистов показывает, что сначала важно подчеркнуть *позитивные* цели этого метода программирования. Мы уже рассмотрели один *негативный* аспект структурного программирования (по крайней мере, с точки зрения среднего программиста), т. е. исключение GO TO. В следующем разделе мы увидим, что обычно оно состоит в ряде ограничений и правил программирования, которые обеспечивают соответствие программы, очень строгому образцу, исключая тем самым бессистемность, неудобочитаемость и запутанность, которые порождают ошибки и затрудняют тестирование и сопровождение. Это также может показаться нежелательным аспектом, заставляющим программиста выразить свое неудовольствие следующими словами: *«Еще* правила и ограничения в дополнение к стандартам, которые мне уже приходится соблюдать. Программировать стало совсем неинтересно; у меня нет возможности для творчества!» В этом есть элемент истины, хотя обычно страхи слишком преувеличены. В то же время, если мы сможем показать, что вводимые ограничения позволяют программисту удвоить число команд, которые он пишет за день, то это, быть может, вовсе не так уж плохо!

4.2.1. Уменьшение трудностей тестирования

Мотивы, побудившие первоначально Дейкстру к разработке структурного программирования, остаются определяющими и сегодня. С этим подходом связывается одна из немногих надежд на то, что мы когда-либо сможем тестировать большие программы и программные системы тщательно и в полном объеме. Имеющиеся здесь трудности хорошо известны многим программистам и упоминались в разных разделах этой книги. Превосходное обсуждение проблем тестирования программ читатель может найти также в журнале *Software Engineering* [13] и *Software Engineering Techniques* [14]. Здесь мы ограничимся следующим кратким изложением этих проблем:

1. Трудоемкость и стоимость тестирования больших программ возрастает экспоненциально с увеличением их размеров. А. д'Агапаев в журнале *Software Engineering* [13] сообщает, что стоимость проверки изменения в большой системе с разделением времени почти в 100 раз больше стоимости внесения этого изменения. «Хуже того,— нам известно, что наши программы становятся больше и несомненно будут продолжать увеличиваться и в дальнейшем».

2. Кажется, что ошибки всегда остаются в больших системах, особенно в таких, которые требуют постоянного сопровождения, усовершенствования и других изменений. В журнале *Software Engineering Techniques* [14] М. Гопкинс (который вышецитировался как сторонник *сохранения* оператора GO TO) сообщает, что в каждой новой версии ОС Системы IBM/360 имеется около 1000 ошибок и это число остается относительно неизменным. Мой опыт и изучение операционных систем других изготовителей подтверждают это явление; число ошибок в разных системах может быть большим или меньшим, но каждый изготовитель, по-видимому, характеризуется своей константой.

3. Издержки от неиспытанных систем неуклонно возрастают по меретого, как общество возлагает на электронные вычислительные системы все более и более ответственные функции. Методы тестирования, используемые при создании систем автоматического управления и других «критических» систем, в большинстве случаев оказываются достаточными, поскольку многие из этих систем сравнительно невелики. Здесь достаточно даже доморощенных приемов тестирования, чтобы выявить и устранить все ошибки. В случае больших систем коллективного пользования, работающих в реальном масштабе времени (под большими мы понимаем системы, включающие *десятки* взаимосвязанных машин, базы данных емкостью до *миллиарда* символов, к которым всегда имеется непосредственный доступ, *сотни* программистов, создающих программы, содержащие более *миллиона* строк кодов), аналитики систем и программисты весьма серьезно опасаются, что какая-нибудь невыявленная ошибка может быть причиной огромных человеческих жертв, денежных потерь или неуправляемости систем принятия решений (например, систем управления воздушным транспортом).

На что же мы можем рассчитывать в будущем, если учесть все эти трудности. Некоторые методы тестирования, рассматриваемые в гл. 7, помогут устранить бессистемность и недостаток организации, которые сопутствуют, кажется, большинству испытаний. Некоторые исследования, проводимые в настоящее время в этой области, смогут помочь руководителям более точно оценивать необходимые для проведения испытаний ресурсы времени и денежных средств. Различные пакеты моделирующих программ и генераторов тестовых данных смогут до некоторой степени облегчить автоматизацию тестовых процедур. И все же эти средства по своей природе носят частный характер. Программист выбирает то, что он считает хорошим тестовым случаем, и проводит его испытание. После того как опробовано достаточное количество случаев, программист прекращает эту работу и заявляет, что программа испытана. Как за-* мечает Дейкстра в журнале *Software Engineering Techniques* [14], «тестирование доказывает наличие ошибок, но не их отсутствие».

Другой возможный путь спасения лежит в области автоматиче-

ского доказательства правильности программ. Несколько исследователей изучали проблему автоматического доказательства правильности произвольной программы; некоторые из полученных ими результатов можно найти в работах [15—17]. Эти результаты, однако, не обещают каких-либо практических приложений в ближайшем будущем. В одном из случаев, выполненных Хоором, доказательство правильности программы, содержащей 12 операторов, включает восемнадцать лемм [17]. Более того, все еще сохраняется такое чувство, что число операторов «доказательства» может возрастать с увеличением длины проверяемой программы быстрее, чем по линейному закону. Одной из причин разработки Дейкстрой структурного программирования было то, что автоматическое доказательство программ, отвечающих некоторому структурному образцу, может быть значительно упрощено. Представляется, что так оно и есть на самом деле, однако до сих пор мы не имеем ни одного практического метода получения строгого доказательства правильности программ, записанных в *какой-либо* форме.

Вместе с тем программы, написанные на основе принципов структурного программирования с использованием подхода «сверху вниз» (о чем будет сказано в разд. 4.5), оказываются более простыми в тестировании, опирающемся на частные методы. Дейкстра, например, приводит следующее описание успешного тестирования операционной системы для системы мультипрограммирования «THE», спроектированной по нисходящей схеме (однако, без какой-либо ссылки на структурное программирование): «...Мы установили, что проектирование модифицированной системы мультипрограммирования можно вести так, что правильность ее логической структуры будет поддаваться доказательству, а ее реализация будет допускать всестороннее тестирование. Единственный вид ошибок, обнаруженных при тестировании, составили тривиальные ошибки кодирования (встречавшиеся с частотой около одной ошибки на 500 команд), которые, естественно, было легко исправить. К настоящему моменту тестирование еще не завершено, но имеется гарантия, что законченная система не будет содержать ошибок.

Опыт проекта фирмы IBM оказался в тестировании столь же успешным: в основной логической структуре системы тестированием не было обнаружено ошибок, а те немногие ошибки, которые были выявлены, оказались тривиальными и легко фиксируемыми».

4.2.2. Более высокая производительность программистов

Хотя это может казаться очевидным, стоит заметить, что облегчение тестирования обычно повышает производительность программиста, т. е., используя этот подход, программист может написать большее число отлаженных команд программы в день. Многими исследова-

ниями установлены следующие приближенные оценки производительности программистов (см., например, статью Арона в журнале *Software Engineering Techniques* [18]):

Характер программирования	Число операторов, отлаженных за день
Операционные системы	1-3
Системные программы-компиляторы	5—10
Прикладные программы	10—15
Структурированные программы	35—65
Супер программисты	50—200

Хотя существующие экспериментальные подтверждения еще не собраны, есть основания полагать, что в области систем реального времени (операционные системы, системы управления процессами и т. д.) повышение производительности окажется даже большим, чем в программировании прикладных задач в модельном времени. Приведенные выше полные энтузиазма высказывания Дейкстры, кажется, подтверждают эту мысль.

Повышение производительности более существенно, чем может это казаться на первый взгляд. Если каждый программист может сделать вдвое больше, то для выполнения данного проекта требуется лишь половина программистов; часто это позволяет уменьшить число уровней в организационной структуре управления и значительно улучшить психологическую атмосферу и общение между программистами. Среди N программистов отдельныйпрограммистчувствует себя как малая спица в большом колесе, среди $N/2$ программистов он может почувствовать себя более важной составляющей и работать с большим упорством и научиться значительно большему в искусстве (а может науке?) программирования.

4.2.3. Ясность и читабельность программ

Структурное программирование обладает тем дополнительным преимуществом, что повышает *читабельность* программ (этот его аспект обсуждается более подробно в гл. 5). Поведение многих неструктурированных программ часто ближе к броуновскому движению, чем к сколько-нибудь организованному процессу. Всякая попытка прочесть листинг приводит человека в отчаяние тем, что в такой программе обычно исполняются несколько операторов, после чего управление передается в некоторую точку несколькими страницами ниже, где исполняются еще несколько операторов и управление снова передается в какую-то случайную точку, там испол-

няются еще какие-то операторы и т. д. После нескольких таких передач читатель забывает, с чего все началось, и теряет ход мысли.

Структурированным программам, напротив, свойственна тенденция к последовательной организациии и исполнению. Еще в большей степени это справедливо в отношении программ, отвечающих определенным форматам и соглашениям, выделяющим вложенные уровни циклов и операторов IF-THEN. Как заметили программисты проекта IBM:

> С исключением операторов GO TO программу стало возможным читать сверху донизу без перерывов передачами управлений, так что одним взглядом можно определить условия, необходимые для модификации того или иного блока программы.

Как мы увидим в дальнейшем, правила структурного программирования часто сопровождаются некоторыми соглашениями относительно форматов и использованием некоторых идей модульного программирования, которое мы обсудили в предыдущей главе. Так, в случае проекта IBM были предприняты энергичные меры по ограничению размера каждого модуля пятьюдесятью операторами.

4.2.4. Эффективность

Один из наиболее распространенных доводов против структурного программирования сводится к утверждению, что оно приводит к менее эффективным программам. Повышенное внимание уделяется тому обстоятельству, что использование операторов вызова подпрограмм как альтернативы операторов GO TO увеличивает время ЦП, требуемое для прохождения всей программы, и в ряде случаев связано с необходимостью значительного увеличения памяти. В других ситуациях программисту проще повторить небольшие фрагменты программы (предпочитая этот прием обращению к «общему» разделу программы), дабы не нарушать правил структурного программирования. Хотя обычно это не приводит к увеличению времени ЦП, ясно, что на это требуется дополнительная память.

В настоящее время признано, что программы, написанные на языках высокого уровня, таких, как Алгол, ПЛ/1 и Кобол, потенциально оказываются *более* эффективными при использовании принципов структурного программирования. В конечном счете эффективность программы, написанной на языке высокого уровня, зависит от качества объектного кода, формируемого компилятором с такого языка (если, конечно, не принимать во внимание эффективность или неэффективность программы, заложенные в ней как внутренние свойства проекта, что часто оказывается значительно важнее рассматриваемых деталей кодирования). Создатели компиляторов начали обращать внимание на то, что *глобальная оптимизация* (т. е. совместная оптимизация нескольких операторных выражений высокого уровня описания или оптимизация всей программы в целом)

значительно проще в случае структурированных программ, так как управление такими программами может анализироваться автоматическими средствами. Мое личное мнение таково, что достигаемый при оптимизации выигрыш значительно превышает все виды неэффективностей, внутренне присущих принципам структурного программирования. И если этого не удается достичь средствами современных компиляторов, то, безусловно, можно будет добиться компиляторами нескольких будущих лет.

4.3. Теория и методы структурного программирования

Как мы уже видели, понятие структурного программирования представляет собой некоторые *принципы* написания программ в соответствии с набором жестких правил и имеет целью облегчение процессов тестирования, повышение производительности программистов и улучшение читабельности результирующей программы. Прежде всего к *методам* структурного программирования относится отказ от оператора GO TO и замена его рядом других более структурированных операторов передачи управления. Сюда относятся также идеи нисходящего проектирования, рассмотренные нами в гл. 2, а также ряд других менее важных ограничений и соглашений, касающихся программирования.

В этом разделе более детально будут обсуждены некоторые теоретические и практические аспекты структурного программирования. Мы рассмотрим реализацию этой концепции в таких языках, как Алгол и ПЛ/1; в менее изящных языках программирования типа Фортран и Кобол; в языке ассемблера и в так называемых языках ассемблера высокого уровня. Мы рассмотрим также другие аспекты структурного программирования, которые обычно игнорируются в горячих дебатах по поводу оператора GO TO. Наконец, мы рассмотрим некоторые вопросы, касающиеся эффективности, удобства и т. д., все еще привлекающие внимание ряда специалистов вычислительной науки.

4.3.1. Теоретические основания структурного программирования

Несколькими авторами уже было замечено, что формальные системы теории вычислимости не требуют понятия GO TO. Так, общие рекурсивные функции Клини [19],системы Поста [20], алгоритмы Маркова [21]илямбда-исчислениеЧёрча [22]строятсябез использования механизма GO TO. Однако, как отметил Ливенворс [23], большинство языков программирования, базирующихся на этих системах, например Лисп, ISWIM, COMIT и Снобол, *все же* включают оператор GO TO. Существенным в этом наблюдении является то, что оператор GO TO, очевидно, не обязателен для вычисления

всех вычислимых функций (и, таким образом, для решения всех практических вычислительных задач), но он представляется удобным.

В области, более практической, ряд исследователей пытались найти решение проблемы написания программ таким образом, чтобы их правильность поддавалась доказательству, что мы уже отметили в начале главы. В согласии с прагматическими целями, провозглашенными Дейкстрой, эти усилия были приложены в направлении анализа программ, организованных по нисходящей схеме, или «слоисто». В этой схеме вся программа (или система) сначала рассматривается как независимый «вызываемый» модуль (и действительно, часто она именно так и функционирует, поскольку всякая прикладная программа *вызывается* операционной системой как подпрограмма, и при этом она отрабатывает *возврат* управления операционной системе по окончании или сбою). На следующем этапе проектирования исходная (уровня 0) программа расчленяется на подпрограммы модулей уровня 1; эти последние подвергаются декомпозиции на подмодули уровня 2; процесс декомпозиции продолжается до тех пор, пока проектировщик не придет к настолько малым составляющим блокам, которые могут быть легко закодированы.

При тестировании всей программы важно иметь возможность определить поведение подмодулей уровня k независимо от конкретных условий на более высоком уровне. Это позволило бы нам доказывать правильность подмодулей уровня $(k+1)$ независимо от смысла, придаваемого им на k-м шаге декомпозиции. Это в свою очередь с необходимостью приводит к требованию, чтобы всякий модуль проектировался с единственным входом и единственным выходом; что опять ведет к представлению о программе как множестве вложенных модулей, каждый из которых имеет один вход и один выход.

В классической работе Бома и Джакопини [24] было показано, что такая структура может быть реализована в языке, включающем только две основные управляющие конструкции, фактическая реализация которых может быть различной в разных языках программирования, как мы это увидим в разд. 4.3.2. Принцип, изложенный в статье Бома и Джакопини, иногда упоминаемый как «структурная теорема», имеет фундаментальное значение и составляет основу большинства *реализаций,* которые мы рассмотрим ниже! Доказательство того, что *всякая* реальная программа может быть построена с использованием лишь двух основных управляющих конструкций, представляется, таким образом, весьма важным, однако оно столь длинное и сложное, что здесь обсуждаться не будет. Представляет некоторый исторический интерес то, что в оригинале статья была напечатана на итальянском языке в 1965 г.; в английском переводе она была опубликована в Соединенных Штатах в мае 1966 г. и долго

не привлекала к себе внимания — так же как многие ранние статьи Дейкстры и других авторов до тех пор, пока в начале 70-х гг. структурное программирование не стало модным.

По Бому и Джакопини для построения программы нам требуется три основных составляющих блока:

1. Функциональный блок.
2. Конструкция обобщенного цикла.
3. Конструкция принятия двоичного, или дихотомического, решения.

Функциональный блок, показанный на рис. 4.1, можно представлять как отдельный вычислительный оператор (или команду

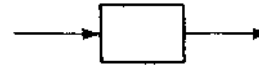


Рис. 4.1. Функциональный блок.

в машинном коде) *или как любую другую реальную последовательность вычислений с единственным входом и единственным выходом,* как в подпрограмме. Так, функциональным блоком может быть команда «считать в сумматор» в языке ассемблера, оператор MOVE в Коболе, или типичное вычисляемое выражение в Фортране. Организация цикла, показанная на рис. 4.2,а, в литературе часто упоминается как элемент DO-WHILE. Конструкция принятия двоичного решения показана на рис. 4.2,6; по очевидным причинам его часто называют элементом IF-THEN-ELSE.

Заметим, что конструкции, показанные на рис. 4.2,а и 6, могут сами рассматриваться как функциональные блоки, поскольку они обладают только одним входом и одним выходом. Таким образом, мы можем ввести *преобразование* операции цикла в функциональный блок, как это показано на рис. *4.3,а,* и в последующем рассматривать всякий такой оператор цикла эквивалентом (несколько более сложного) функционального блока. Аналогично мы можем ввести преобразование конструкции принятия решения вида, представленного на рис. 4.2,6, к функциональному блоку, как это показано на рис. 4.3,6. Наконец, мы можем привести всякую последовательность функциональных элементов к одному функциональному элементу, как это показано на рис. 4.3, *в.*

Всякая программа, составленная из функциональных блоков, операторов цикла и элементов IF-THEN-ELSE (а в соответствии с результатами Бома иДжакопини программа может быть построена с использованием только этих элементов), поддается последовательному преобразованию, как это иллюстрируется рис. 4.3, *а — в,* к единственному функциональному блоку. Как отмечает Вулф [11], эта последовательность преобразований может быть использована как средство понимания программы и доказательства ее правильности. В то же время *обратная* последовательность преобразований

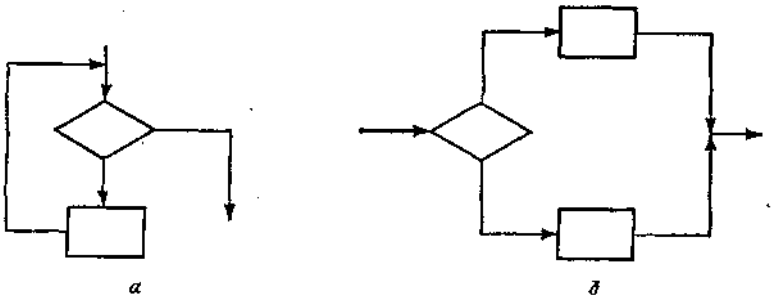


Рис. 4.2. Две логические конструкции.
а — циклическая конструкция; *б* — конструкция IF-THEN-ELSE.

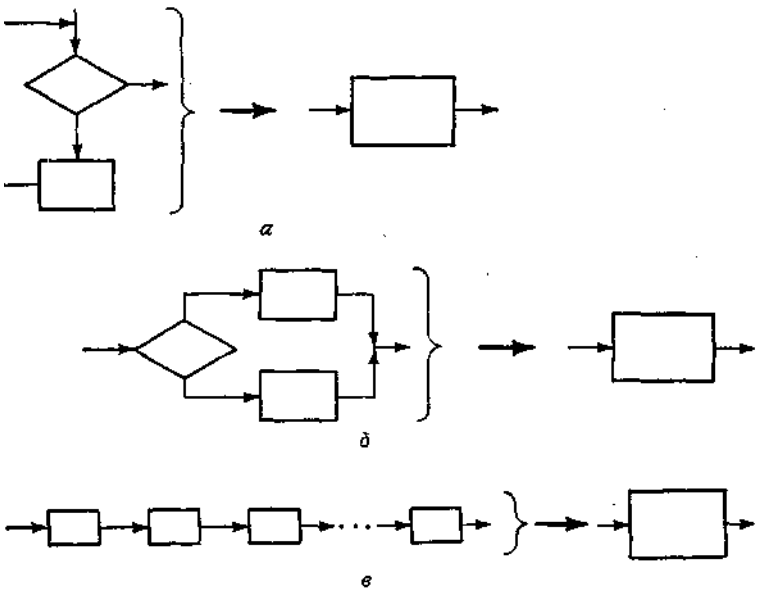


Рис. 4.3. Преобразования.

может быть использована в процессе проектирования программы по нисходящей схеме, т. е. исходя из единственного функционального блока, который постепенно раскрывается в сложную структуру основных элементов.

Отметим также взаимосвязь между этой идеей «вложенной структуры» и идеей модульности, рассмотренной в предыдущей главе. Программа, построенная путем применения приведенных выше преобразований, является модульной в строгом смысле слова. Любой из функциональных элементов может быть заменен эквивалентным, не вызывая никаких последствий в остальной части

программы. Еще важнее то, что преобразования Бома—Джакопини могут быть применены к исходной программе, разбивая ее тем самым на меньшие модули, которые в свою очередь могут быть разбиты на еще меньшие модули и т. д. В соответствии с этой схемой процесс преобразований может продолжаться до тех пор, пока мы не достигнем уровня «атомарных» модулей т. е. отдельных вычислительных операторов, операторов IF-THEN-EISE и элементарных циклов типа DO-WHILE.

В этом, таким образом, состоит существенное различие в подходах структурного программирования и модульного программирования. Попытки применить модульное программирование обычно характеризуются хорошим началом: программист решает разбить большую программу на модули. Часто, однако, он не пытается разбить каждый из полученных модулей на меньшие модули. В результате получается программа, содержащая главный модуль и небольшое число больших (например, в несколько сот операторов) модулей первого уровня, которые не легко разделить на составляющие. Отметим важность этого обстоятельства: если модули не поддаются дальнейшему разбиению, то они должны рассматриваться как неделимые структуры, а это означает, что тестирование, отладка, сопровождение или понимание таких модулей будут затруднены в связи с необходимостью целостного восприятия больших блоков программного кода. Ситуация обычно осложняется еще тем, что эти модули, введенные неформально в соответствии с общими представлениями модульного программирования, часто оказываются *зависимыми* один от другого — они изменяют логику друг друга, пользуются общими областями рабочей памяти и т. д.

4.3.2. Реализация структурного программирования

Теоретическая основа структурного программирования допускает реализацию его принципов на многих современных языках программирования. Соответствующие правила очень просты: все операции в программе должны представлять собой либо непосредственно исполняемые в линейном порядке выражения (например, обычные арифметические операторы), либо одну из следующих трех управляющих конструкций:

1. Вызовы процедур, подпрограмм и функций — любое допустимое обращение к замкнутой подпрограмме с одним входом и одним выходом. Заметим, что подпрограммы *не* являются абсолютно необходимым условием возможности реализации структурного программирования. Однако, рассуждая в том же духе, мы могли бы сказать, что всякая программа может быть реализована в форме машины Тьюринга и кое-кто в наше время пытается это сделать!
2. Вложенные на произвольную глубину операторы IF-THEN-ELSE.

3. Некоторые циклические структуры. Чаще всего в таком качестве используются конструкции DO-WHILE, входящие в Алгол и ПЛ/1, однако приемлема и конструкция Кобола PERFORM-UNTIL.

Хотя перечисленных средств достаточно для построения произвольной программы для ЭВМ, в ряде организаций находят удобным использовать дополнительно некоторые их «расширения». Ниже приводятся некоторые из наиболее распространенных расширений.

1. Конструкция CASE. Встречаетсявомногихнеофициальных версиях Алгола 60 и в некоторых современных языках типа Алгол может быть представлена в нескольких формах; простая реализация, принятая в языке B5500 Алгол фирмы Burroughs, имеет вид

```
CASE  GLOP  OF
   BEGIN
   ОПЕРАТОР1:
   ОПЕРАТОР2:


   ОПЕРАТОР]';
   END.
```

Если переменная GLOP принимает значение *i,* то будет исполнен *i-й* оператор и затем управление будет передано оператору, следующему за конструкцией CASE. Операторы, входящие в состав конструкции CASE, сами, конечно, могут быть сложными функциональными элементами. Заметим, что сама конструкция CASE может быть представлена в форме «черного ящика», как это показано на рис. 4.4. Следует отметить также, что конструкция CASE

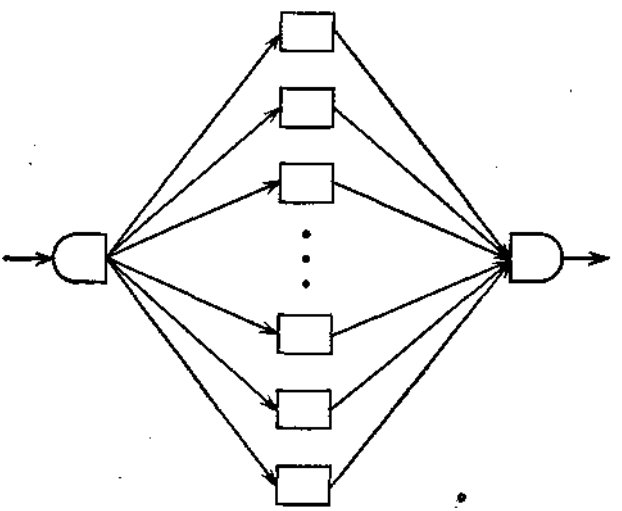


Рис. 4.4. Конструкция CASE.

```
9
t
        PERFORM CASE-SECTION.


CASE-SECTION.
        GO TO A, B, C, D DEPENDING ON I.
A.      -

        GO TO CASE-EXIT.
B.      -

        GO TO CASE-EXIT,
C.      -

        GO TO CASE-EXIT,
D.      -

        GO TO CASE-EXIT,
CASE-EXIT.
        EXIT.
```

Рис. 4.5. Моделирование конструкции CASE в Коболе.

```
                GO TO (1, 2, 3, ...) I
10              CONTINUE


1       —

        GO TO 10
2       —

        GO TO 10
3       -

        GO TO 10


        и т. д.
```

Рис. 4.6. Моделирование конструкции CASE в Фортране.

может быть реализована в Коболе с помощью оператора GO TO DEPENDING ON, как это иллюстрируется на рис. 4.5, с помощью вычисляемого оператора GO TO в Фортране, как показано это на рис. 4.6, и с помощью множества различных средств в языке ассемблера.

2. Дополнительные конструкции организации цикла. Удобно, например, иметь оператор цикла, в котором *сначала* исполняется функциональный блок, а *затем* проверяется необходимость его повторения, как это показано на рис. 4.7. Эта форма существует

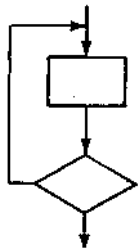


Рис. 4.7. Альтернативная форма основной логической конструкции.

в некоторых реализациях Алгола, а множество более простых вариантов — в Фортране и языке ассемблера. Заметим, что цикл, показанный на рис. 4.7, представляется традиционной формой черного ящика, введенной первоначально Бомом и Джакопини. Точно ьак же кратные операторы цикла, имеющиеся в большинстве языков (например, оператор DO в Фортране и PERFORM-VARYING в Коболе), при *соответствующем их применении* представляются в виде черных ящиков.

3. Подпрограммы со многими входами и многими выходами. Некоторые программисты предпочитают, например, кодировать функции SIN и COS в одном модуле с различными точками входа; аналогично, многие программисты имеют обыкновение вводить в каждой подпрограмме «нормальный» выход и выход «по ошибке». Если этим пользоваться с осторожностью, то условия, связываемые с понятием черного ящика, могут не нарушаться; на практике, однако, они часто нарушаются.

4. Жесткие ограничения на применение операторов GO TO. В некоторых организациях программисту, например, разрешается использовать передачу управления только вперед по программе; в других — применяют те или иные вариации этого специального правила. Одна из наиболее интересных состоит в следующем: программисту (который работает на Коболе) предписывается оформлять свою программу в виде Кобол-секции, в которую можно войти только по оператору PERFORM и из которой можно выйти, только выполнив оператор EXIT последнего параграфа секции. При внешнем обращении нельзя адресоваться к тому или иному абзацу

внутри секции или передавать управление в произвольную точку секции; аналогично программисту запрещено использование операторов GO TO для передачи управления из его программы-секции. В дополнение к этому размер секции ограничивается приблизительно 100 операторами (что напоминает нам ограничения, рассмотренные в разд. 3.1). Во всех остальных случаях, однако, программисту разрешено применять оператор GO TO по своему усмотре-

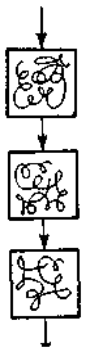


Рис. 4.8. Ограниченное употребление операторов GO TO. Программный код внутри верхнего модуля (обозначен квадратом) может быть достаточно сложным, однако в нем не допускаются передачи управления с выходом из модуля.

нию, что, вероятно, может иметь результатом лишь локальную (в пределах секции) хаотическую организацию программы, как это иллюстрируется рис. 4.8. Хотя в рамках чистой теории структурного программирования это можно рассматривать как некоторый компромисс, он может оказаться необходимым, когда речь идет о таких негибких языках программирования, как Кобол и Фортран.

Могут быть предложены и многие другие модификации и компромиссы основных представлений структурного программирования, и их, возможно, *предложат,* когда в организациях, разрабатывающих программы, освоят основные принципы этого подхода. Как было отмечено, многие из описанных компромиссных решений не нарушают исходных принципов структурирования Бома и Джакопини, связываемых с понятием «черного ящика»; другие решения *определенно* нарушают эти принципы, и их применение должно быть ограничено только исключительными обстоятельствами, например жесткими требованиями используемого языка программирования или экстремальными условиями обеспечения эффективности. Важно, чтобы был определен некоторый удобный набор соглашений по реализации принципов структурного программирования, включающий, возможно, ряд из числа охарактеризованных выше компромиссов и расширений, которым затем должен быть *придан* статус стандартных правил программирования.

4.3.3. Обращение неструктурированных программ в структурированные

Из результатов Бома и Джакопини следует, что *всякая* реальная программа может быть построена из ряда основных составляющих блоков, рассмотренных в предыдущем разделе. Многие программисты, однако, обнаруживают, что при попытке писать программы в соответствии с таким взглядом они сталкиваются с большими' трудностями. Простейшие прикладные задачи могут, конечно, быть запрограммированы без оператора GO TO (который для многих программистов остается контрольным признаком того, было ли должным образом применено структурное программирование). Те, кому повезло программировать на ПЛ/1, Алголе или других достаточно развитых языках, отмечают, что им редко приходится нарушать принципы структурного программирования; однако те, которым приходится программировать на Коболе, Фортране или языке ассемблера, часто считают, что освоение структурного программирования составляет для них даже большую трудность, чем изучение нового языка программирования.

Многие трудности объясняются тем фактом, что структурное программирование требует иного подхода к проектированию и пониманию программ. До некоторой степени освоению этого подхода способствует нисходящая методология, рассмотренная в гл. 2. Он воспринимается как вполне естественный теми, кто имеет опыт в программировании на языках типа Алгол; другим, однако, он кажется весьма неожиданным. Попростуговоря, многие из нас постоянно программировали в бессистемной хаотической манере, и совершенно не достаточно узнать, что существуют теоремы, доказывающие возможность структурного программирования.

К счастью, существуют некоторые методы, которые могут облегчить переход к структурному программированию, хотя первоначально эти методы были созданы для решения несколько иной задачи. Ряд ученых, работающих в вычислительной науке, поставили следующий вопрос: можно ли обратить в структурированную программу некоторую произвольную программу, которая первоначально не была написана в соответствии с правилами структурного программирования? В общем случае ответ звучит негативно: произвольная программа не может быть преобразована в структурированную программу, *которая реализует тот же самый алгоритм, построена на тех же самых элементах и не использует дополнительных переменных;* см. Кнут и Флойд [25] и Ашкрофт и Манна [26], где обсуждается этот результат. Однако мы можем выполнить такое преобразование, если согласимся на введение в программный код некоторых дополнительных фрагментов или использование вспомогательных переменных состояния или управляющих признаков. Это может быть достигнуто применением трех известных методов: дубли-

рования кодов программы, введения переменной состояния и метода булевых признаков.

Ниже рассматривается каждый из этих методов. Важно, однако, заметить, что эти методы предполагают соответствующее их использование. Если мы возьмем плохо спроектированную неструктурированную программу и преобразуем ее в соответствии с перечисленными выше методами, то в результате мы получим плохо спроектированную, не использующую операторы GO TO структурированную программу, понять и отладить которую, вероятно, столь же трудно, как и исходную. И действительно, хотя некоторые исследования имеют своей целью создание автоматического транслятора неструктурированных программ в структурированные, не ясно, согласится ли какая-нибудь организация допустить такое разрушительное нападение на свои библиотеки существующих программ!

В то же время не следует забывать, что наш традиционный подход к проектированию программ чаще всего оказывается неструктурным; имея это в виду, стоит заметить, что три названные стратегии преобразования программ во многих случаях помогут нам овладеть методами *проектирования* и *понимания* программ, которые будут отличаться более структурированным характером. Таким образом, мы определенно *не* рекомендуем вам писать программы, используя ваши прежние методы, и затем обращать их в структуризованные без оператора GO TO программы с помощью названных выше приемов. Вместо этого вам следует попытаться с их помощью развить в себе структурное мышление при решении задач программирования. Мы должны еще раз подчеркнуть, чтоэтомувогромной степени способствует применение нисходящего проектирования прсграмм.

Дублирование кодов. Рассмотрим программу, блок-схема которой показана на рис. 4.9. Под каждым блоком этой схемы понимается группа операторов соответствующего абзаца Кобол-программы, блок главной программы на Фортране или блок операторов на языке ассемблера. Стрелки, соединяющие различные элементы, представляют в данном случае оператор GO TO.

В настоящем виде программа не является структуризованной; каждый блок схемы *не* удовлетворяет описанному выше требованию «один вход — один выход». В очень конкретных практических случаяхкаждый программист сталкивался с ошибками в программах подобной структуры; выражаясь неформально, в данном случае наши затруднения объясняются тем, что «передача управления в область общего пользования» чревата многими осложнениями (обратите внимание на то, что структура программы, показанная на рис. 4.9, удовлетворяет требованию ограничить использование оператора GO TO только для «передачи управления вперед по программе», которое некоторыми организациями принимается в качестве компромиссной версии структурного программирования).

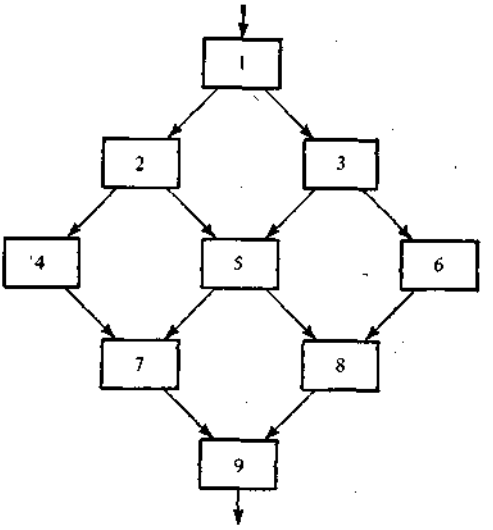


Рис. 4.9. Пример неструктурированной программы.

Как пример трудностей, которые встречаются в такой программе, рассмотрим попытку разобраться в работе модуля 5 (такие задачи обычно возникают при тестировании, отладке и сопровождении). Мы можем предположить, что в типичной программе правильное функционирование модуля 5 зависит от значений различных признаков, переключателей и других переменных, которые были ранее должным образом обеспечены. Эти признаки и переменные могли быть правильно заданы при входе в модуль 5 из модуля 2 и не обеспечены при входе из модуля 3. Другими словами, мы не можем судить о правильности функционирования модуля 5, не зная поведения модулей 2 и 3 (и, таким образом, по индукции — модуля 1). Мы не можем установить правильность работы модуля 5, не зная контекста, в котором он исполняется. Эта задача оказывается еще более сложной в случае модулей 7 и 8, так как существует несколько путей, которыми можно попасть в эти модули. Таким образом, если модуль 5 «не срабатывает», то может оказаться, что очень трудно определить, произошло ли это из-за ошибки в самом модуле 5 или из-за ошибки в одном из предшествующих ему модулей.

Существует еще одна трудность, которая довольно часто встречается при таком программировании. Нередко мы замечаем, что в модуле 5 содержатся программные коды, которые совершенно необходимы, если мы вошли в этот модуль из модуля 2, но не являются таковыми и вполне безобидны, если войти туда из модуля 3. Если программа действительно работает таким образом, то нас не

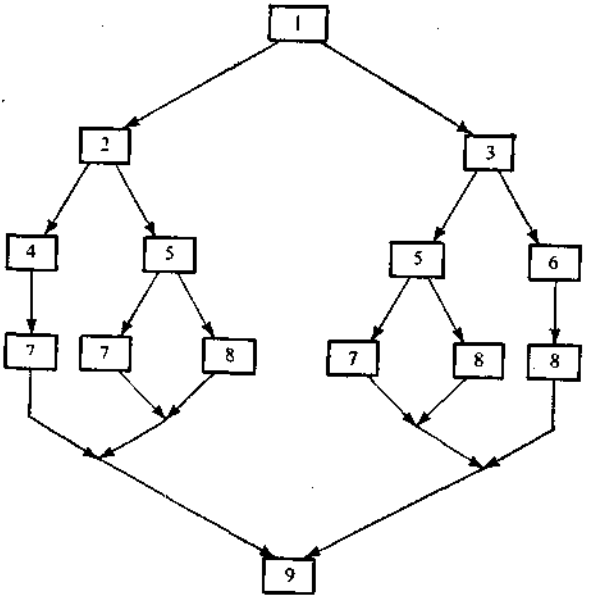


Рис. 4.10. Преобразованная форма схемы рис. 4.9.

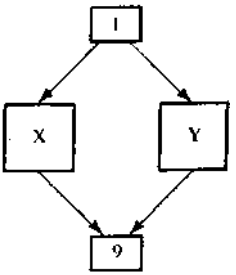


Рис. 4.11а. Упрощенное представление схемы по рис. 4.10.

ожидают никакие неприятности; к сожалению, мы обнаруживаем, что названные коды необходимы при входе из модуля 2 и *разрушительны,* если мы входим из модуля 3. Таковы опасности обращения в область общих кодов, и в тот или иной периодсвоей работы каждый программист, вероятно, испытал эти неприятности.

К счастью, эта задача решается сравнительно просто. Чтобы получить структурированную программу, мы воспользуемся *дублированием* тех модулей, в которые можно войти из нескольких мест. В рассмотренном примере, который мы связываем с рис. 4.9, это приводит к преобразованной блок-схеме, показанной нарис.4.10. С несколько иной точки зрения рассмотрим исходную программу

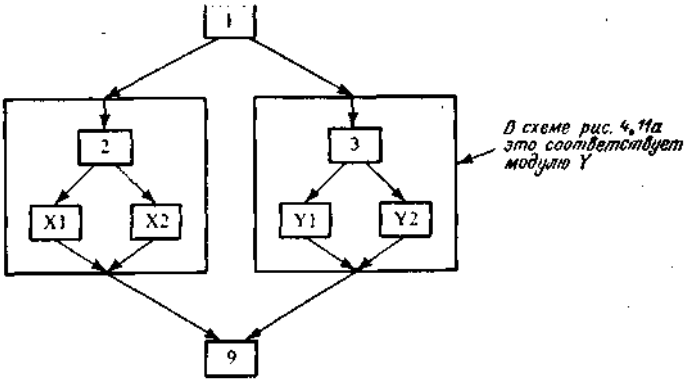


Рис. 4.11б. Более подробное представление схемы по рис, 4.11а.

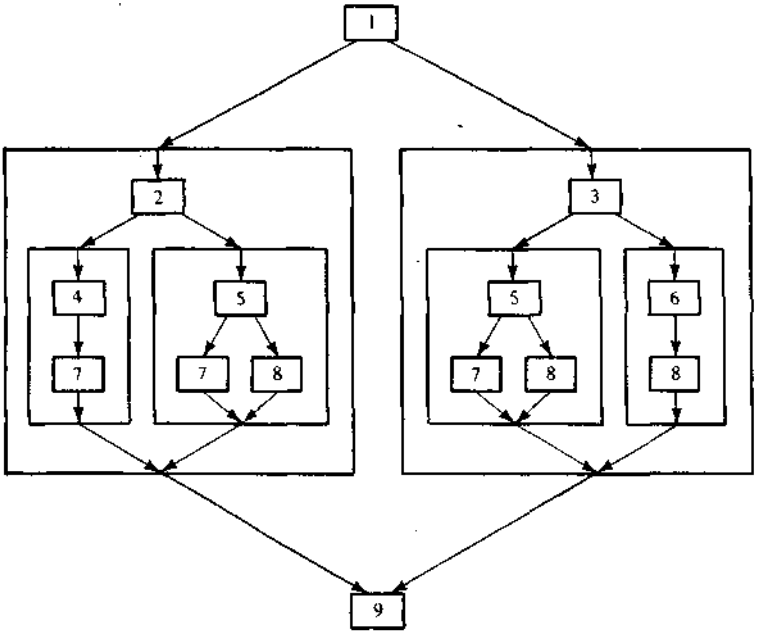


Рис. 4.11в. Более подробное представление схемы по рис. 4.11б.

как простую конструкцию типа IF-THEN-ELSE, показанную на рис.4.11а;онаможетбытьрасширенадоструктуры,изображенной нарис.4.11б,окончательновсяпрограммаможетбытьизображена, как это показано на рис. 4.11в. Одним из сильнейших доводов в пользу метода «дублирования кодов» является то, что наши *представления* и процесс *проектирования,* связанные с эво-

люцией из состояния рис. 4.11а через рис. 4.11б к состоянию рис. 4.11в, существенно проще, чем в случае решетчатой структуры рис. 4.9, и тем самым в меньшей степени чреваты ошибками.

Не вызывает серьезных сомнений и то, что описанный метод применим к любой программе, имеющей структуру типа показанной на рис. 4.9. Мы, однако, должны отметить, что программа крайне проста — в ней нет циклов. Метод дублирования кодов обычно не может быть применен к программам с циклами, скорее всего он применим лишь к структурам типа решеток и сетей. Для преобразования программ, содержащих циклы, предназначаются два других метода, которые нам предстоит рассмотреть ниже. Стоит, однако, напомнить, что составляющие многих более сложных программ имеют сетевую структуру и что с этими фрагментами программ следует обращаться так же, как это иллюстрируется рис. 4.10.

Метод дублирования кодов имеет очевидный недостаток: он требует больше памяти, чем исходный неструктурный подход. Однако очень часто оказывается, что дублируемые модули (например, различные блоки в схеме рис. 4.9) в такой программе содержат всего по два-три оператора. В таком случае дублирование кодов — приемлемая плата за возможность получить распадающуюся на уровни структуру, подобную приведенной на рис. 4.11а — 4.11в. Если же модули состоят из значительного объема кодов (например, 50 и более операторов исходной программы), то ясно, что решением задачи может быть введение вызываемых подпрограмм. При этом очень важно, чтобы они были организованы как *формальные* подпрограммы с формальными параметрами, так что их правильность может быть установлена вне зависимости от контекста, в котором они используются. В таком подходе мы используем многократное обращение к одной и той же подпрограмме, что означает относительно небольшой дополнительный расход памяти.

Интересно проследить процесс построения блок-схемы, приведенной на рис. 4.9. Какпоказано на рис. 4.12,а, программист начинает с кодирования модулей 1, 2 и 3. Выполнив эту работу, он приступает к раскрытию модуля 2, а именно к кодированию модулей 4 и 5, как это показано на рис. 4.12, *б.* Далее программист начинает расписывать модуль 3, что иллюстрируется рис. 4.12, *в,* и именно здесь он понимает, что один из подмодулей (помеченный, как 5 А) очень похож на модуль, который он составил при раскрытии модуля 2. На самом деле такое представление часто возникает еще *до того,* как происходит фактическое кодирование; когда программист еще только *думает* о составе модулей 2 и 3, он осознает, что модули 5 и 5 А (которые пока являются лишь расплывчатыми мысленными образами) довольно похожи друг на друга и что их следует объединить; отсюда схема, представленная рис. 4.9. Во многих случаях, однако, модули, задуманные как 5 и 5 А, похожи друг на друга, но не идентичны и их *следует оставить как разные*

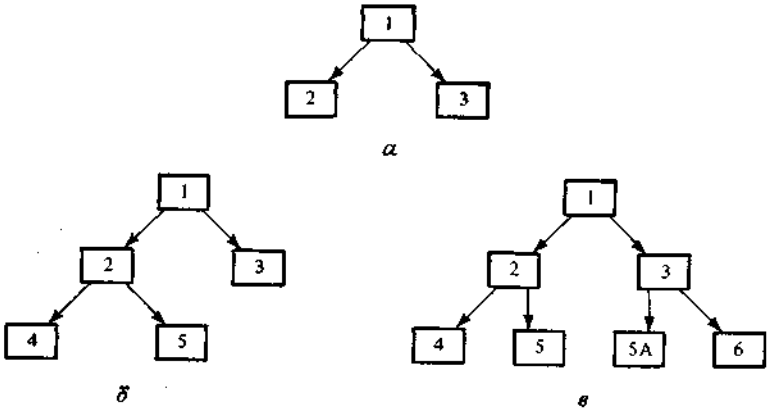


Рис. 4.12. *а* — начало разработки программы по схеме рис. 4.9; *б* и *в* — продолжение разработки программы по схеме рис. 4.9.

модули, в особенности если их реализация составляет лишь несколько строк кодов.

Многие программисты настаивают на том, что структурированная версия программы, представленная рис. 4.10, не обязательно будет проще или тем более легче в отладке. Они замечают, например, что в модуле 5 схемы рис. 4.10 ошибка может возникнуть точно так же, как она может возникнуть в модуле 5 исходной схемы рис. 4.9. Естественно, что это так. Возможность внести ошибки в наши программы существует вне зависимости от того, как мы их структурируем. В то же время мы можем утверждать, что программа, представленная рис. 4.10, логически менее сложна и что поэтому меньше вероятность того, что в нашу логику вкралась ошибка.

Более того, мы можем настаивать на том, что программисту следует анализировать логику программы по нисходящей схеме, и это способствовало бы устранению ошибок, прежде чем они будут фактически возникать в программном коде. Так, программист мог бы проверить часть программы, показанную на рис. 4.11а, и убедиться, что в ней нет ошибок.. Затем он мог бы рассмотреть более низкие уровни в соответствии с рис. 4.11б и убедиться в том, что они правильные. Наконец, он мог бы проверить самые низкие уровни схемы рис. 4.11в и убедиться в том, что вся программа правильная.

Стоит также заметить, что причиной многих ошибок в неструктурированных программах (например, таких, как исходная версия по рис. 4.9) являются «неформальные» интерфейсы, например программист не дает формального определения данных, которые должны быть засланы в модуль 5 расположенными выше модулями 2 и 3. В структурированном подходе, показанном на рис. 4.10, эти интерфейсы формализуются, например, когда определяют модуль 5

какформальную подпрограмму с формальными параметрами. В этом случае, если модуль 5 «отказывает» в процессе исполнения программы, мы смогли бы установить, является ли причина внутренней (т. е. ошибка в самом модуле 5) или внешней (т. е. передача ошибочной информации через интерфейс).

Наконец, нам следует рассмотреть еще одно возражение, выдвигаемое некоторыми программистами: было отмечено, что вся программа, представленная рис. 4.9, является «черным ящиком», т. е. имеет один вход (в модуль 1) и один выход (из модуля 9). Предполагая, что программа довольно мала, например все девять модулей составляют в сумме от двадцати до тридцати строк кодов, многие программисты сомневаются в необходимости преобразования программы в структурированную форму рис. 4.10. Хотя это верно, нужно иметь в виду следующие два момента:

1. Программа, показанная на рис. 4.9, очевидно, является малой программой. Логические схемы *реальных* программ часто намного сложнее, т. е. решетчатые структуры нередко охватывают несколько сотен или даже несколько тысяч строк программных кодов.

2. Даже в том случае, когда логическая структура программы по рис. 4.9 легко прослеживается (как утверждают некоторые программисты), мы считаем основной целью этого раздела показать, что всякая программная ситуация, в том числе и решетчатые структуры, может быть разрешена в терминах основных элементов Бома и Джакопини. Таким образом, мы могли бы утверждать следующее: если программа, представленная схемой рис. 4.9, и является разумной (что, во всяком случае, остается спорным), то ее организация не исключает, что программа будет ошибочно применяться посредственными программистами (или даже хорошими программистами, если у них выдался неудачный день!). А поскольку применение основных элементов Бома и Джакопини делает решетчатые структуры необязательными, нет оснований допускать возможность их неверного использования.

Метод введения переменной состояния. Другой метод преобразования неструктурированных программ был предложен Ашкрофтом и Манной в [26]. Он представляет собой незначительную вариацию метода переменной состояния, используемого многими программистами в организации их программ (см., например, обсуждение в разд. 6.3.2). В этом методе интересно то, что он применим к любым программам (в частности, содержащим циклы и другие сложные конструкции) и что он допускает автоматическое применение.

Процесс преобразования состоит из пяти шагов; он иллюстрируется примером программы, показанной на рис. 4.13.

1. Каждому блоку неструктурированной схемы приписывается номер; заметим, что на схеме рис. 4.13 это уже сделано. Способ

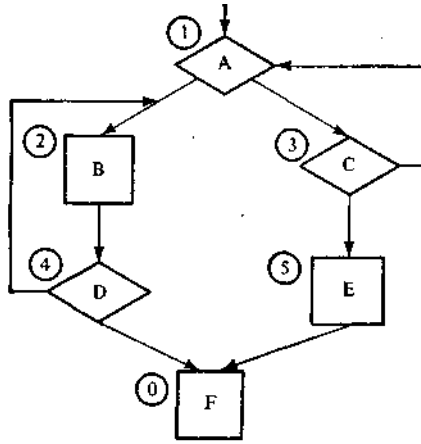


Рис. 4.13. Пример, иллюстрирующий метод Ашкрофта — Манны.

присваивания номеров совершенно произвольный, но обычно принимают соглашение обозначать номером 1 первый исполняемый блок программы и номером 0 — последний исполняемый блок.

2. В программу вводится новая переменная; для нашей цели обращения программы в структурированную нам требуется переменная целого типа. Имя переменной произвольное; так, в нашем примере новая переменная обозначена через *i*.

3. Функциональные блоки неструктурированной схемы заменяются функциональными блоками, которые выполняют те же самые вычисления и, кроме того, присваивают переменной *i* целое значение, идентифицирующее номер *блока-преемника* в исходной схеме. Так, блок *B* на рис. 4.13 во всех случаях, очевидно, передает управление блоку *D*. Так как мы пометили блок *B* цифрой *2,* то новую его версию обозначим через 2'; этот новый блок выполняет тот же процесс, что и *B,* но в дополнение присваивает переменной *i* значение *4*.

4. Логические блоки исходной схемы преобразуются таким же образом. Если решением логического блока является «истина», то переменной *i* присваивается номер блока-преемника исходной схемы, отвечающего этому решению; соответственно решению «ложь» отвечает значение переменной *i,* равное номеру блока-преемника, определяемого в исходной схеме решением «ложь». В схеме рис 4.13, например, блок *1* выполняет проверку *A,* определяющую выбор перехода на блок *2* или блок *3*. Мы заменим блок *1* блоком *1'*, который исполняет проверку *A* и устанавливает значением переменной *i* целое *2,* если условие А удовлетворяется и целое *3* — в противном случае.

5. Теперь мы перестроим всю блок-схему, придав ей форму, показанную на рис. 4.14. Начальным значением переменной *i* выбрано

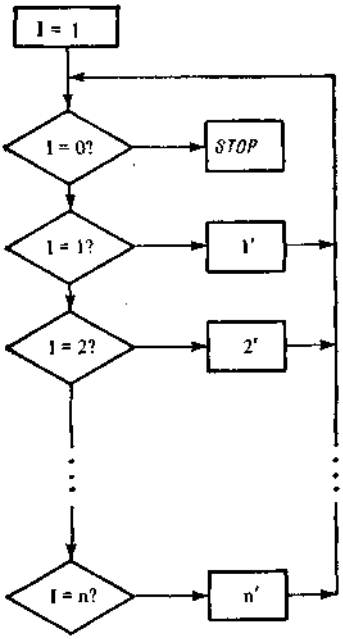
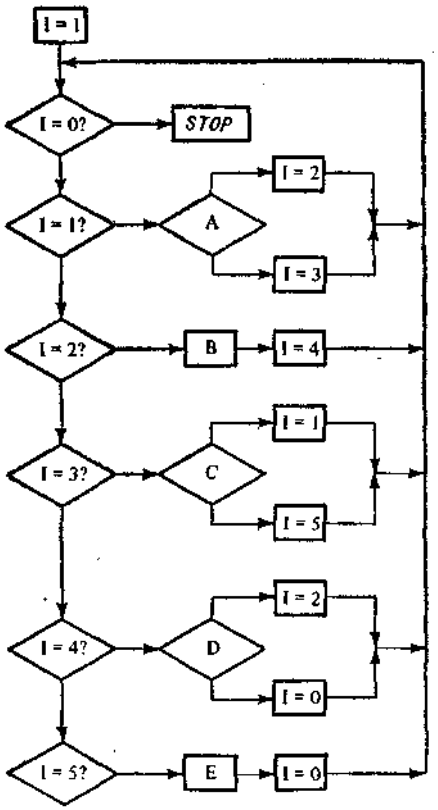


Рис. 4.14. Преобразование по методу Ашкрофта — Манны.

в соответствии с принятым выше соглашением целое 1. Затем мы последовательно выполняем опрос значения переменной *i*, исполняем соответствующее действие, повторяем опрос *i* и т. д. Пример, показанный на рис. 4.13, может, таким образом, быть представлен схемой рис. 4.15.

Этот процесс обращения отличается наглядностью и четкостью и привлекает многих программистов; у иных, однако, по ряду причин он вызывает возражение. Одно из наиболее распространенных возражений связывают с тем, что этим методом преобразования разрушается форма и топология исходной блок-схемы; например, схема рис. 4.15 не имеет никакого сходства с исходной блок-схемой, представленной рис. 4.13. Это возражение часто формулируется несколько более сильно: программист заявляет, что схему рис. 4.13 легче понимать и что схема, показанная на рис. 4.15, не обладает необходимой наглядностью в части принимаемых решений.

В какой-то степени эти возражения могут быть справедливыми. Стоит, однако, заметить, что схема по рис. 4.13 довольно проста, ведь она умещается всего на одной странице! Кроме того, хотя ей и присуща внутренняя неструктурированность, схема представляется «черным ящиком», т. е. имеет лишь один вход и один выход. Программируя в *реальных* условиях, мы можем иметь дело с блок-



Рис, 4.15. Структурированная форма схемы по рис. 4.13.

схемами, которые *не* помещаются на одной странице; более того, нам могут встретиться схемы с многими входами и многими выходами. Преобразованная введением переменной состояния форма обладает тем преимуществом, что может быть неограниченно продолжена (например, вместо шести состояний, характеризующих схему рис. 4.13, мы могли бы рассмотреть пример с шестьюдесятью состояниями), не усложняя при этом общего подхода.

Еще одно достоинство метода состоит в ясности, которая может быть не сразу замечена. Подход, связанный с введением переменной состояния, помогает программисту в документировании программы, что может быть крайне полезным для программиста сопровождения. Каждому блоку исходной схемы соответствует определенное *состояние* программы, т. е." ситуация, в которой либо проверяется логическое условие, либо выполняется некоторая обработка. Эти состояния могут быть документированы в четкой и компактной форме; еще более важно то, что переходы между

состояниями могут быть представлены в табличной форме, как мы увидим из примера, приведенного в разд. 6.3.5.

Метод введения переменной состояния обладает еще некоторым преимуществом в отладке, на которое часто не обращают внимания. Если программа не выполняется должным образом,- то довольно просто *трассировать* переменную состояния (например, переменную *i* в нашем случае). Это должно дать программисту довольно ясное представление о ходе управления программой.

Наконец, возникает вопрос об эффективности. Многие программисты на основе схемы рис. 4.15 делают вывод, что реализация метода введения переменной состояния предполагает использование вложенных конструкций IF-THEN-ELSE. Хотя этот способ возможен, более вероятна реализация с использованием сочетания конструкции DO-WHILE (или PERFORM-UNTIL, или его эквивалента в других языка) и конструкции CASE (или вычисляемого оператора GO TO в Фортране, или GO TO DEPENDING ON в Коболе и т. д.), которые после компиляции программы в машинном коде по своему действию будут равносильны употреблению переменной состояния в качестве индекса в «векторе передач управления». Совершенно верно, что каждый функциональный блок исходной программы дополняется операцией присваивания значения переменной состояния и что значение переменной состояния должно опрашиваться после исполнения каждого блока, однако это сравнительно малые издержки.

Метод булевого признака. Существует еще один метод программирования, который может быть использован для обращения в структурированную форму программ, содержащих циклы; в этом методе требуется введение в программу некоторого *признака* (FLAG), хотя не исключено, что он уже имеется. Начальное значение признака задается в некоторой точке выше цикла; конструкциями типа DO-WHILE или PERFORM-UNTIL осуществляется управление

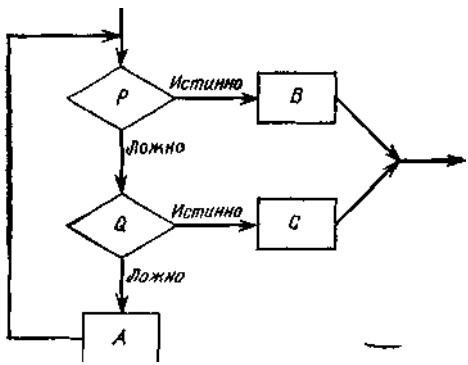


Рис. 4.16а. Неструктурированный цикл.

циклом до тех пор, пока названный признак сохраняет заданное значение; и наконец, некоторыми условиями внутри цикла определяется момент смены значения признака. Таким образом, программа представляется в форме:

```
FLAG=0

WHILE  FLAG = 0 DO
       BEGIN

       •

        IF  X = Y  THEN  FLAG=I

       END
```

или в какой-нибудь другой эквивалентной форме в иных языках программирования.

Эта процедура может быть до некоторой степени обобщена. Одной из наиболее распространенных форм неструктурированного цикла является форма, показанная на рис. 4.16а. Обратите внимание на то, что в ней имеется только одна точка входа и два различных выхода. Введением нового признака мы можем привести эту программу к виду, показанному на рис. 4.16б. Хотя в блок-схеме это выглядит, возможно, несколько неуклюже, такое приведение довольно просто программируется использованием переменной FLAG и оператора DO-WHILE.

Иногда преобразование программы не требует введения в нее признака. Например, неструктурированный цикл схемы рис. 4.16а можно было бы преобразовать, как показано на рис. 4.16в. Это решение намного яснее и изящнее, чем то, что приведено на рис. 4.16б; построение, однако, становится слишком громоздким, если цикл содержит *несколько* различных выходов вместо двух, как в нашем исходном примере по рис. 4.16а.

В качестве примера рассмотрим упрощенную версию программы обновления основного файла, показанную на рис. 4.17а. Традиционный способ кодирования такого алгоритма показан на рис.4.17б, более структурированный вариант представляет рис.4.17в. Обратите внимание на то, что в этом примере использованы метод

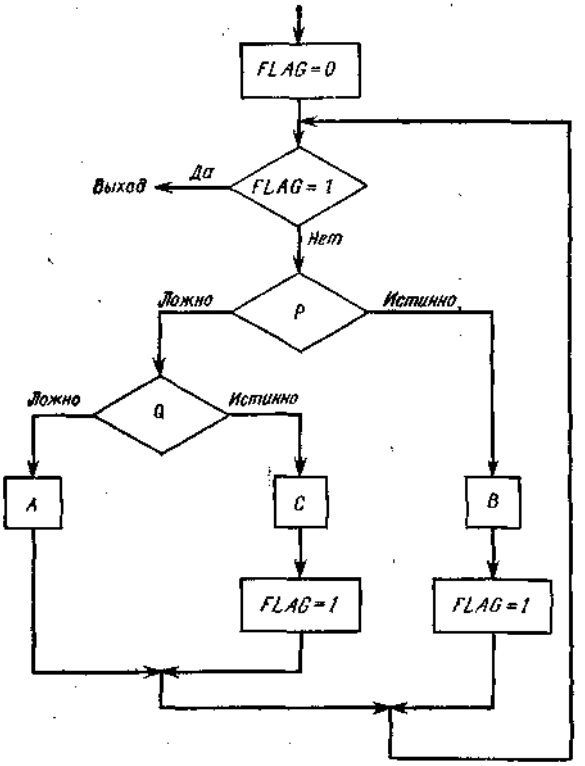


Рис. 4.16б. Структурированный вариант схемы по рис. 4.16а.

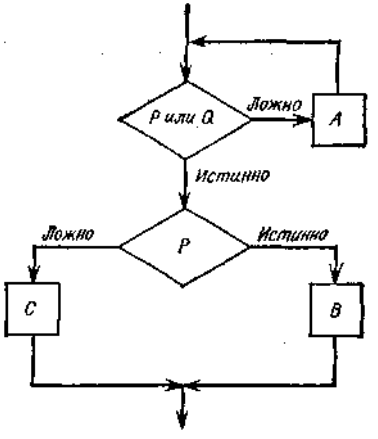


Рис. 4.16в. Другой вариант структурирования схемы по рис. 4.16а.

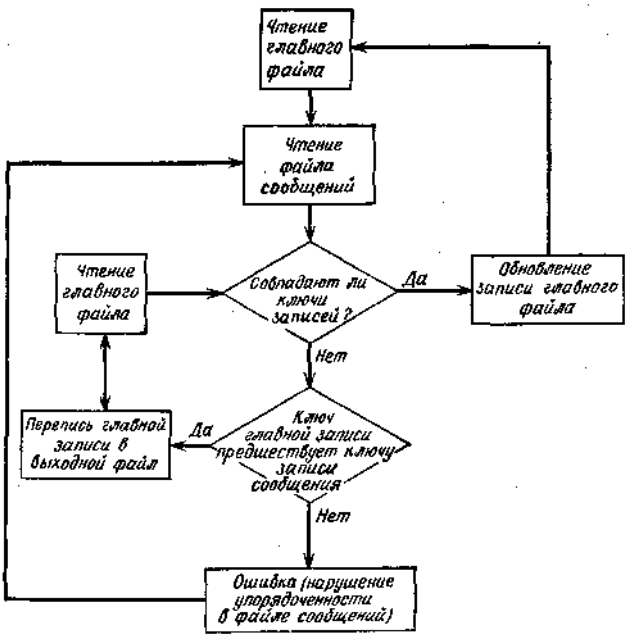


Рис. 4.17а. Упрощенный вариант алгоритма обновления главного файла.

Примечания

1. Иа этой блок-схеме не представлены процессы отработки конца файла.
2. Алгоритмом по этой блок-схеме не допускается возможность сопоставления нескольких записей файла сообщений одной записи главного файла.
3. Алгоритмом по этой блок-схеме не допускается внесение новых записей (файла сообщений) в главный файл.

```
X-MASTER.
    READ MASTER.
X-TRANSACTION.
    READ TRANSACTION,
X-COMPARE.
    IF M = T
        PERFORM UPDATE
        GO TO X-MASTER
    IF M < T
        PERFORM WRITE-MASTER
        PERFORM READ-MASTER
        GO TO X-COMPARE.
    PERFORM ERROR-MESSAGE.
    GO TO X-TRANSACTION
```

Рис. 4.17б. Неструктурированный вариант программы по рис. 4.16.

```
        PERFORM  READ-MASTER.
        PERFORM  READ-TRANS.
        PERFORM  APPLY-TRANS  UNTIL  M = T = 9999.


    APPLY-TRANS
        IF  M = T
            PERFORM  UPDATE-MASTER
            PERFORM  READ-MASTER
            PERFORM  READ-TRANS
        ELSE
            IF M < T
               PERFORM  WRITE-MASTER
               PERFORM  READ-MASTER
            ELSE
               PERFORM  ERROR-MESSAGE
               PERFORM  READ-TRANS.
        EXIT.
    READ-MASTER.
        READ  MASTER  AT  END  MOVE  9999  TO  M.
READ-TRANS.
        READ  TRANS  AT  END  MOVE  9999  TO  T.
```

Рис. 4.17в. Более структурированный вариант программы по рис. 4.16.

введения булева признака *и* рассмотренный выше способ дублирования кодов.

Метод введения булева признака дает нам, по-видимому, лучший пример ситуации, в которой структурное программирование требует совершенно иного мышления в проектировании программ. После некоторого опыта применения этого метода многие программисты находят такое мышление вполне естественным, и лишь первые несколько случаев кажутся довольно трудными. Мы должны подчеркнуть еще раз, что примеры, выбранные для иллюстрации приведенных методов преобразования программ, в какой-то степени могут вводить в заблуждение: каждый из примеров (рис. 4.9, 4.13 и 4.16) отвечает крайне простому алгоритму, и эти алгоритмы *уже представляются в виде черных ящиков*. Причина, по которой они были выбраны, состоит в том, что в первоначальной своей форме их схемы не поддаются декомпозиции на *меньшие* черные ящики.

4.3.4. Применение структурного программирования в общепринятых языках

Принимая во внимание, что в структурном программировании существенно требование замены оператора GO TO конструкциями DO-WHILE и IF-THEN-ELSE, нам следует теперь задаться вопросом, располагают ли соответствующими средствами общепринятые языки программирования. В следующих разделах мы сосредоточим внимание на нескольких группах языков.

Алгол и ПЛ/1. Алгол и ПЛ/1 являются языками, которые допускают самую простую реализацию идей структурного программирования. Одной из основных причин этого является наличие в них «блочной структуры», которая позволяет программисту группировать вместе несколько отдельных операторов и затем обращаться с ними, как с *составным* оператором. Вместе с гибким оператором IF-THEN-ELSE в рассматриваемых языках это позволяет строить такие конструкции, как

```
IF BE1 THEN
   BEGIN
   ОПЕРАТОР1;
   ОПЕРАТОР2;
   IF BE2 THEN ОПЕРАТОР3;
   ОПЕРАТОР4;
   END
ELSE
   IF BE3 THEN
   BEGIN
   ОПЕРАТОР1;
   ОПЕРАТОР2;
   END
ELSE
   BEGIN
   ОПЕРАТОР!;
   IF BE4 THEN ОПЕРАТОР2 ELSE ОПЕРАТОР3;
   ОПЕРАТОР4;
   END:
```

Другим мощным оператором, входящим в состав этих двух языков, является конструкция DO-WHILE. С ее помощью программист может выполнять один или более операторов до тех пор, пока булево выражение, определяемое предложением WHILE, истинно. В Алголе имеются варианты этого основного средства, позволяющие располагать проверку значения булевой переменной либо *перед,* либо *после* кодов исполняемого блока. Таким образом, программист может использовать такую организацию цикла, в которой входящие в его состав операторы не исполняются вовсе (если условия окон-

чания цикла выполняются уже при входе в цикл), или такую, в которой они исполняются хотя бы раз, прежде чем будут проверены условия окончания цикла. Как мы отметили выше, конструкция CASE, входящая в состав некоторых версий Алгола, оказывается весьма полезной при построении описанных в этой главе типов структур, в особенности в методе переменной состояния, рассмотренной ранее.

По-видимому, важнейшими с точки зрения структурного программирования достоинствами этих языков являются точность и краткость их синтаксиса. Это становится особенно заметно при сравнении операторов IF-THEN-ELSE языка типа Алгола с таким же оператором языка типа Кобола — блочная структура обычно позволяет исключить неоднозначность, встречающуюся часто в Коболе при использовании *вложенных* конструкций IE-THEN-ELSE. Аналогично фора оператора DO-WHILE в Алголе и ПЛ/1 позволяет кодировать цикл по принципу *«внутрь»,* а не *«наружу»* (как в случае, например, конструкции PERFORM-UNTIL в языке Кобол 10).

Кобол. Хотя и не столь изящный и формализованный, как Алгол и ПЛ/1, язык Кобол располагает достаточными средствами для того, чтобы реализация структурного программирования получалось довольно удобной. Все три основные составляющие элемента поддаются в Коболе непосредственной реализации: функциональные блоки, конструкции IF-THEN-ELSE и DO-WHILE.

Конструкции IF-THEN-ELSE могут вкладываться одна в другую, хотя большинство программистов на Коболе, вероятно, предупреждены своими руководителями о том, что этими конструкциями не следует пользоваться. Частично это объясняется трудностью чтения последовательности кодов, подобной, например, следующей:

```
IF I = 1
   SUBTRACT 1 FROM J
   IF J = 2
       MOVE A TO B
ELSE
   IF  I=3
       MOVE C TO D
```

Тут нет ясности, относится ли предложение ELSE к оператору IF J = 2 или к начальному оператору I = 1. Следует подчеркнуть, что неясность состоит не в том, как обработает эту последовательность компилятор, поскольку мы предполагаем, что действия компилятора предсказуемы и известны (хотя, быть может, различаются у разных поставщиков!). *Мы имеем в виду, что компилятор часто будет делать не то, что намеревался сделать программист.*

Иногда эту трудность можно разрешить здравым употреблением предложения Кобола ELSE NEXT SENTENCE, но порой и это средство оказывается недостаточным. На худой конец, мы всегда можем прибегнуть к оператору PERFORM; так, предыдущий пример может быть закодирован следующим образом:

```
IF I = 1
        PERFORM X
ELSE
        PERFORM Y
```

где модуль X содержит следующие операторы:

```
X.
        SUBTRACT 1 FROM J.
        IF J = 2 MOVE A TO B.
        EXIT,
```

и подобным же образом записывается содержание модуля Y. Заметьте, однако, насколько громоздко выглядит этот подход в сравнении с соответствующей реализацией на Алголе, достаточно громоздко, к сожалению, чтобы вызвать у некоторых программистов на Коболе нежелание обращаться к структурному программированию.

Аналогично конструкция DO-WHILE допускает в Коболе непосредственную реализацию с помощью оператора PERFORM-UNTIL. Общая форма этого оператора, к сожалению, довольно неудобна для реализации циклов, так как строится по принципу «наружу», т. е. в форме

```
PERFORM ABC UNTIL X = Y
```

причем абзац или секция ABC должны располагаться где-то в другой части программы. Мы предпочли бы йметь возможность писать что-нибудь вроде

```
UNTIL X = Y PERFORM
    MOVE A TO B
    MOVE C TO D
    ADD 1 TO X
```

Заметим, наконец, что понятие блочной структуры не присутствует в этом, языке в явной форме, но может быть сымитировано объединением нескольких простейших операторов (например, MOVE ADD или COMPUTE) в одно-предложение. Хотя далеко не столь формальный, как его эквиваленты в языках Алгол и ПЛ/1 (обстоятельство, которое иногда приводит к некоторым неприятностям), этот механизм выполняет преобразование нескольких отдельных функциональных блоков в единственный функциональный блок.

Фортран. Фортран является, по-видимому, единственным известным языком высокого уровня (разве что наряду с BASIC), ко-

торый *не* отвечает идеям структурного программирования. Это объясняется тремя основными причинами:

1. В языке Фортран отсутствует блочная структура в современном представлении этого понятия. То есть отсутствует возможность объединять вместе несколько операторов и обращаться с ними как с *одним,* оператором. Это немедленно приводит программиста к необходимости «перескакивать» через группы операторов. Хотя эти переходы по оператору GO TO сами по себе особого зла не представляют, они открывают программисту (соблазнительную) возможность начать строить сложные структуры типа показанной на рис. 4.9.

2. В языке Фортран отсутствует возможность вкладывать один в другой операторы IF-THEN-ELSE. По мнению автора, отсутствие сильной конструкции IF является наибольшей слабостью Фортрана.

3. Хотя в Фортране *имеется* конструкция DO, она не столь гибка, как аналогичные средства в Коболе, Алголе и ПЛ/1. Использование в Фортране оператора DO затруднительно для каких-нибудь целей, кроме организации итерационных процессов (что, конечно, является обязательным условием научных приложений). В языке нет аналогов операторов DO-WHILE или PERFORM-UNTIL.

Хотя эти положения содержат резкое суждение о языке Фортран, они вовсе не означают, что на Фортране нельзя писать структурированные программы. Напомним, что *действительной* целью структурного программирования является последовательная декомпозиция программы на более мелкие дискретные элементы. Нет никаких причин, по которым этого нельзя было бы сделать *с помощью* оператора GO TO на Фортране, *если бы только была гарантия, что он небудетиспользоватьсяпрограммистамидлякаких-либоиныхцелей.* При этих условиях конструкция IF-THEN-ELSE, например, может быть реализована в Фортране, как это показано на рис. 4.18а; аналогично может быть реализована конструкция DO-WHILE, показанная на рис. 4.186. В альтернативном подходе можно было бы создать «супер-Фортран» — язык, включающий операторы IF-THEN-ELSE и DO-WHILE и обладающий блочной структурой. В этом случае некоторый препроцессор осуществлял бы транслирование этих конструкций в *стандартные* операторы Фортрана, которые все мы знаем и любим. Такой подход уже был предпринят по крайней мере одной вычислительной организацией в Европе, одной компанией по разработке математического обеспечения в Соединенных Штатах (см., например, объявление относительно препроцессора языка Ифтран в январском выпуске журнала *Datamation* за 1974 г.) и некоторыми пользователями системы UNIVAC 1108 (см. работу [40]).

В настоящее время, однако, Фортран представляется довольно слабым средством для реализации концепций структурного программирования. Ту дисциплину, которую пришлось бы навязать программистам, чтобы обеспечить структурированное употребление операторов GO TO путем использования лишь форматов, показан-

```
      IF (булево выражение) GO TO 10

      Программные коды для случая „ложь"

      GO TO 20

10

      Программные коды для случая „истина"

20    CONTINUE
```

Рис. 4.18а. Реализация конструкции IF-THEN-ELSE на Фортране.

```
10    IF (булево выражение) GO TO 20

      Тело цикла

      GO TO 10
20    CONTINUE
```

Рис. 4.18б. Реализация конструкции DO-WHILE наФортране.

ных на рис. 4.18а и рис. 4.18б, было бы трудно, если вообще возможно, контролировать.

Языки системных разработок. В последние несколько лет были созданы языки ассемблера высокого уровня, позволяющие системным программистам использовать преимущества языков высокого уровня (удобные средства управления и организации циклов, контроля ошибок, структурыданных), сохраняя при этом эффективность

и управляемость языка ассемблера «низкого уровня». В настоящее время тремя наиболее известными языками этого типа являются ПЛ/С, BLISS и ПЛ/360. BLISS был создан группой исследователей Университета им. Карнеги — Меллона и описан в литературе [29]; он был использован как инструмент в нескольких разработках, выполненных в духе структурного программирования. ПЛ/360 был создан Виртом для Системы IBM/360 и описан в работе [30]; в нем делается попытка предоставить программисту гибкость и организацию ПЛ/1, соединенные с эффективностью и управляемостью языка ассемблера Системы IBM/360. ПЛ/С (язык программирования для систем) был разработан на фирме IBM в качестве языка программирования систем математического обеспечения на таких машинах с виртуальной памятью, как 370/158. В момент написания этой книги фирма IBM, владея этим языком на правах собственности, не согласилась сообщить подробнее о его свойствах.

Эти два языка и другие им подобные (в качестве дополнительных примеров можно привести язык B5500 ESPOL фирмы Burroughs и Project MAC EPL, использованный в разработке системы GE-645 MULTICS) обычно вполне согласуются с идеями структурного программирования. На самом деле одна из основных целей создания этих языков состояла в том, чтобы исключить определенные трудности отладки и сопровождения, столь свойственные программам на языке ассемблера, и, работая в этом направлении, системные программисты начали осознавать, что если они действительно желают создавать системы, свободные от ошибок, то им необходимо соблюдать некоторую организацию их программ. Интересно отметить, кстати, что язык BLISS не содержит явной формы оператора GO TO, в то время как в языке ПЛ/360 программисту разрешается использовать безусловную передачу управления, если он того пожелает.

Существует один потенциальный недостаток, свойственный языкам системных разработок, который не имеет отношения к структурному программированию, но который, между прочим, следует кратко упомянуть. Реализация названных языков почти всегда связана с конкретной машиной, что не позволяет создавать с их помощью программу «на все машины». Например, BLISS былсоздан для машины типа PDP-10 и позволяет программисту интерактивно пользоваться системой команд и архитектурой этой машины. Языки ПЛ/360 и ПЛ/С, конечно, в той же мере приспособлены для машин Систем IBM/360 и IBM/ 370. Это, кажется, не очень беспокоит системных программистов, которые часто отмечают, что их продукция — операционные системы, компиляторы, управляющие пакеты баз данных, и т. д.— никогда не рассчитана «на все машины». Уникальность *всякой* программы обычно довольно преувеличена (почему бы нам не пожелать использовать в PDP-10 некоторые элементы алгоритма оптимизации компилятора Фортран H Системы IBM/360?), и хочется верить, что когда-нибудь мы будем иметь язык

описания и разработки систем, в котором характеристики машинного уровня описания будут отделены от характеристик верхнего уровня.

Язык ассемблера. В заключение нам следует упомянуть язык ассемблера как возможное средство реализации структурного программирования. Сам по себе язык ассемблера характеризуется теми же недостатками, что и Фортран:.по своей природе он не обладает такими вещами, как конструкция IF-THEN-ELSE и DO-WHILE или блочная структура (хотя в некоторых из современных микропрограммируемых машин все возможно!). Тем не менее, добавляя *макрооператоры,* язык ассемблера можно рассматривать как приемлемое средство для реализации идей структурного программирования. Большинство языков ассемблера больших машин (и даже отдельных мини-ЭВМ) позволяют программисту строить макрокоманды, эквивалентные конструкциям IF-THEN-ELSE и DO-WHILE, и макрокоманды BEGIN-END блочной структуры.

Следует напомнить, что отсутствие в языках ассемблера конструкции IF-THEN и т. д. не обязательно исключает возможность структурного программирования. Как мы отметили выше при обсуждении Фортрана, единственно, что здесь требуется,— строгая дисциплина со стороны программиста (или его руководителя) в части использования операций безусловной передачи управления только для реализации тех конструкций, которые естественным образом содержатся в других языках структурного программирования.

4.4. Другие аспекты структурного программирования

В большинстве разработок, использовавших структурное программирование как средство создания программ, применялись и некоторые другие дополнительные правила, ограничения и соглашения относительно программирования. Не являясь *обязательными* составляющими теории структурного программирования, они на практике оказываются довольно удобными. Поскольку многие из этих дополнительных соглашений о программировании обсуждаются в других разделах этой книги, здесь мы лишь кратко их упомянем.

1. Использование соответствующих форматов значительно повышает читабельность структурированной программы. Как мы уже видели, в основе структурного программирования лежат главным образом вложенные обращения к подпрограммам, вложенные конструкции IF-THEN и DO-WHILE; для обеспечения читабельности программы эти построения должны быть выделены. В таких языках, как ПЛ/1 это делается автоматически [31]; в языке Лисп когда-то существовал пакет редактирующих программ, называемый КРАСИВАЯПЕЧАТЬ, в других языках программистам, возможно, приходится проделывать эту работу вручную. В разд. 5.2.3 приво-

дится ряд дополнительных рекомендаций по оформлению программ, повышающих их читабельность.

2. Весьма полезен принцип последовательного разбиения программы на все более мелкие фрагменты, которое следует продолжать до тех пор, пока подлежащие дальнейшему разбиению модули или функциональные блоки не станут сравнительно малыми. Оценки номинального размера модуля варьируются от 20 до 200 операторов в зависимости от используемого языка, характера задачи и привычек программирующего коллектива. Ряд примеров использования *размера* модулей как меры модульности программы рассматривается в разд. 3.1.1. Проектом фирмы IBM [10] в структурном программировании было строго рекомендовано программистам корпорации придерживаться размера модулей, близкого к 50 операторам.

3. Для обеспечения целостности каждого модуля (что *является* неотъемлемой частью концепций структурного программирования) ни в одном модуле (или функциональном блоке, если пользоваться терминологией этой главы) не разрешается модификация какого-нибудь другого модуля. В частности, это означает, что нам следует отказаться от таких операторов, как оператор ALTER в Коболе. Этот принцип уже обсуждался в разд. 1.2.2 и обсуждается еще раз в разд. 5.1.4.

4. Для дальнейшего обеспечения целостности каждого модуля следует добиваться независимости его локализованных переменных и памяти временного хранения от всех остальных модулей. Другими словами, не должно быть памяти временного хранения, общей для, нескольких модулей. Поскольку такая практика по опыту автора, многих его друзей, коллег и клиентов представляется причиной многих ошибок программирования, она неоднократно обсуждается в разд. 1.2.2, 3.3.5 и в гл. 8.

5. Для обеспечения систематической реализации концепции структурного программирования в вашей программе необходимо удостовериться в том, что она проектируется по нисходящей схеме; этот предмет мы обсудили в гл. 2.

4.5. Рассмотрение практических вопросов структурного программирования

После того как мы осветили большую часть важнейших аспектов структурного программирования, нам остается коротко рассмотреть вопросы его реализации в повседневной практике программирования. Существует несколько вопросов, которые, кажется, возникают всюду, где обсуждается структурное программирование. Стоит ли оно прилагаемых усилий и дает ли ожидаемые результаты? Позволяет ли оно создавать эффективные программы и во что оно обходится? Насколько оно удобно (т. е. не встречаются ли в про-

граммировании реальные ситуации, которые не поддаются в удобной форме решению методами структурного программирования)? Действительно ли современные языки программирования подходят для такого способа программирования? Все эти вопросы обсуждаются ниже.

4.5.1. Стоит ли структурное программирование прилагаемых усилий?

Найдется ряд серьезных программистов и ученых, работающих в области ЭВМ, которые полагают, что структурное программирование (или принцип исключения «go to», как его иногда называют) рассматривается как некое универсальное средство решения проблем современного программирования. Большое число программистов, посещавших мои курсы по современным методам программирования, после обсуждения структурного программирования пожимали плечами и спрашивали: «Ну и что? Я не думаю, что исключение операторов GO TQ может очень сильно повлиять на продолжительность кодирования, отладки и т. д.» Хотя порой и признают, что структурированная программа больше подходит для формализованногодоказательства ее правильности, большинство программистов (включая автора) полагают, что соответствующие средства не будут доступными и практичными еще в течение нескольких лет. Скептицизм по поводу «магических» свойств исключения оператора GO TO наилучшим образом, вероятно, был выражен в подытоживающем высказывании Гопкинса [12] (который, между прочим, *признает* некоторые выгоды движения, за исключение «go to»). Вот что он заявил по этому поводу: «Один мудрый философ однажды сказал ленивому царю, что в геометрии нет царской дороги. После того как было установлено в конце 50-х гг., что программирование для ЭВМ представляет собой проблему, в 60-х гг. были предприняты поиски царской дороги в программировании. Были испробованы все пути, включая «понимающие» операционные системы, языки высокого уровня, методы управления разработкой, разделение времени, виртуальную память, обучение программистов и пакеты стандартных программ. И хотя каждое из этих средств оказалось полезным, они не решили проблему программирования. Столкнувшись с этой нерешенной проблемой и различая на горизонте несколько хороших идей, иные в наше время питают надежду, что царская дорога лежит в области стиля и что запрет на оператор *go to* все разрешит. Само существование этой полемики и то значение, которое ей придается некоторыми в остальных отношенияхоченьздравомыслящими людьми, являются симптомами какого-то недомогания в среде людей, причастных к ЭВМ. У нас не много новых обнадеживающих идей. Я подозреваю также, что этот спор отражает какоз?то довольно трудно-постигаемое свойство человеческой природы, представление, что

язык обладает магическим действием и что простое произнесение некоторых слов может быть опасным или развращающим. Может быть несчастье в том, что «go to» состоит из четырех букв?»

Это глубокие и хорошо выраженные мысли, и каждому программисту было бы полезно задуматься над ними. Действительно, запрет на «go to» не обернется для нас чудом, и у нас еще будет более чем достаточно проблем в проектировании, кодировании, тестировании и отладке программ для ЭВМ — в особенности по той причине, что наши аппетиты в области больших систем растут, кажется, быстрее, чем наша способность в их надлежащих проектировании и реализации. Тем не менее наша точка зрения в отношении оператора GO TO, или *структурного программирования,* как нам по-прежнему хотелось бы ее называть, такова что оно *действительно* представляет шаг вперед по сравнению с существующим бессистемнымспособом действий. Является ли этот шаг *значительным,* заслуживает литого внимания и той полемики, предметом которых он является несколько последних лет, не станет известным до тех пор, пока мы не обретем большего опыта в его использовании, включая более полные статистические данные о производительности программистов, количестве ошибок, обнаруженных на этапе тестирования и качестве законченной программы. Наиболее значительным в этом смысле проектом до настоящего времени (по крайней мере наиболее значительным среди освещенных в литературе) был проект фирмы IBM для газеты *Нью-Йорк Таймс* [10]. Опыт этого проекта обнаружил повышение производительности почти в 5 раз, и *это* определенно стоит тех возможных усилий, которые требует структурное программирование. Я так думаю, что после некоторой практики в этой области мы еще, возможно, достигнем цели Дейкстры — повышения производительности программирования на порядок величины.

4.5.2. Эффективно ли структурное программирование?

Другая важная претензия, которая высказывается в адрес структурного программирования, связывается с возможным снижением эффективности программ при использовании этого подхода. В частности, многие исследователи (например, Вулф [11]) выражали недовольство тем, что попытки обратить неструктурированную программу в эквивалентную структурированную программу могут привести к увеличению программы и дополнительным потерям времени из-за введения новой переменной. Другие замечают, что структурное программирование было бы совершенно непрактичным на таких языках, как Фортран, так как основу этого подхода составило бы главным образом увеличение числа обращений к стандартным подпрограммам. Аналогичные претензии высказывались также в, отношении Алгола, ПЛ/1, Кобола, языков системных разработок и даже языка ассемблера.

Наиболее выразительно против приписывания особого значения эффективности высказался, вероятно, Вулф [11]:

Во имя эффективности в программировании вычислений было совершено больше пригрешений (причем не всегда ее удавалось достичь), чем по какой-либо другой причине, включая непроходимую глупость. Одним из таких грехов является хаотическое нагромождение передач управления в программе с использованием нескольких фрагментов программного кода общего пользования. Это как раз тот стиль программирования, от которого необходимо отказаться, если только мы намерены создавать безошибочные, понятные и поддающиеся модификации системы.

Я вполне согласен с этим суждением Вулфа. В наши дни больших развитых операционных систем и языков описания высокого уровня понятие «эффективность» приобрело несколько академический смысл; жалобы программистов, работающих на Коболе с операционной системой IBM/360, по поводу недостаточной эффективности программ должны вызывать лишь слабую улыбку. Если бы нас действительно интересовала эффективность, нам бы до сих пор пришлось программировать в восьмеричном цифровом коде без каких-либо операционных систем.

Несмотря на эту проповедь, ряд программистов, продолжает выражать серьезную озабоченность по поводу потенциальной неэффективности, связанной с применением структурного программирования. Здесь уместно привести следующие комментарии:

1. Метод Ашкрофта — Манны, связанный с введением новой переменной для организации циклов и передачи управления в неструктурированной программе, является, как мы уже заметили, относительно недорогим средством.

Присваивание этойпеременной некоторого целого значения потребует всего лишь одну или две команды; *проверка* ее значения и передача управления соответствующему функциональному блоку часто могут быть выполнены с помощью средств, аналогичных оператору CASE в Алголе или вычисляемому оператору GO TO в Фортране.

2. В других ситуациях может потребоваться дублирование программных кодов с целью избежать решетчатой структуры программы, показанной на рис. 4.9. Если при этом возникает необходимость дублировать длинные фрагменты программы, то это могло бы привести к значительной неэффективности (в смысле использования памяти, но не времени ЦП). С другой стороны, если такое дублирование заменено обращением к подпрограмме в виде одного общего функционального блока, то соответствующие издержки могут быть вполне приемлемыми.

3. В большинстве реализаций структурною программирования отводится значительная роль процедуре *вызова подпрограмм* (например, оператором PERFORM в Коболе, оператором CALL в Форт-

ране), что беспокоит многих программистов, руководители которых определенно запрещают пользоваться соответствующими операторами, так как с этим связывают дополнительные потери ресурсов. В некоторых случаях зто беспокойство обосновано. В одной реализации ПЛ/1 машины 360/50 на процедуру вызова требуется 198мкс [32]. Многие версии Кобола и Фортрана также характеризуются большими потерями по той же причине; сам по себе вызов подпрограммы требует исполнения одной команды в машинном коде, однако команды по пересылке аргументов и сохранению значений регистров общего пользования могут быть действительно дорогостоящими. Эти вопросы требуют тщательного рассмотрения.

4. Использование блочной структуры в языках типа Алгол и ПЛ/1 обычно не требует значительных ресурсов, *если только* программист не описывает в блоке локализованных переменных (в последнем случае возникают те самые потери на «пролог — эпилог», которыми страдают процедуры вызова подпрограмм). Если блочная структура используется просто для формирования составных операторов (что представляет ее естественную функцию группировки нескольких операторов) в конструкциях IF-THEN или DO-WHILE, то эффективность программы не должна снижаться.

5. Конструкция IF-THEN-ELSE является потенциально неэффективным оператором многих языков высокого уровня, *особенно* если она используется вложенно. Кобол и ПЛ/1 пользуются в этом отношении дурной славой; подход, используемый в языках для разработки систем, которые неизбежно более близки к уровню машинного кода, не страдают в заметной степени этим недостатком. Так же как затраты на вызов подпрограмм, эти потери заслуживают тщательного изучения программистом.

В общем случае структурированная программа *будет* несколько менее эффективной по сравнению с неструктурированной программой, однако большая экономия, вероятно, может быть достигнута за счет более тщательного проектирования систем. Во всяком случае, представляется, что повышение производительности программиста на порядок стоит некоторых издержек!

4.5.3. Удобно ли структурное программирование?

Еще одна распространенная претензия к структурному программированию касается его *неудобства:* многим программистам кажется, что программирование без оператора GO TO было бы неудобным, трудоемким и громоздким. В большинстве случаев эта жалоба объясняется только привычкой. Если у вас за плечами многолетняя практика неструктурного программирования *с использованием* оператора GO TO, то, естественно, что вы испытываете определенную неловкость, когда пытаетесь представить перспективы своей работы при таком высокоструктурированном подходе. Однако в силу тех же

причин программировавшие на языке ассемблера в конце 50-х гг. усматривали большие неудобства, когда пытались представить перспективу использования Фортрана (и, кто знает, может быть они были правы!).

Пользуясь словами известной коммерческой телепередачи, на эти претензии можно дать только один ответ: «Попробуйте, и вам понравится!» Опыт нескольких групп программистов в этом отношении оказался довольно единообразным: требуется некоторое время, чтобы привыкнуть к принципам структурного мышления и проектирования, однако, освоив эти методы, их воспринимают как не более трудные, чем любая другая форма программирования. Этот процесс обучения весьма упрощается применением методов преобразования Ашкрофта и Манны, описанных в разд. 4.3.3. Лично мне пришлось пройти через неприятный опыт освоения структурного программирования, не располагая возможностями этих методов: мне пришлось открывать их самому. Я также на своем опыте убедился, что обучение некоторым языкам программирования облегчает переход к структурному программированию. Те, кто знаком с Алголом и ПЛ/1, обычно не сталкиваются с трудностями при его изучении. (На самом деле чисто случайно я обнаружил, что первая моя программа, написанная на Алголе, была структурированной! Нужно ли говорить, насколько я собой гордился?) Программирующие на Коболе и Фортране испытывают несколько большие трудности, хотя представляется, что работающие на Коболе охотнее принимают дисциплину, основанную на преимущественном использовании операторов PERFORM и составных операторов и более редком обращении к оператору GO TO. Те, чей опыт программирования на языке ассемблера включает достаточно частое использование *макрокоманд,* также довольно легко обучаются написанию структурированных программ.

Бытует такое представление, что в программировании существует несколько ситуаций, в которых применение стандартных методов структурного программирования оказывается *неудобным.* Вулф [11], например, указывает на необходимость иметь некоторую конструкцию, позволяющую выходить в произвольной точке из цикла или из многократно вложенных подпрограмм. В соответствии с этой необходимостью в языке BLISS, одним из проектировщиков и создателей которого является Вулф, допускается использование глагола LEAVE, позволяющего выйти из любого такого цикла или процедуры. Аналогичные дополнения к основным средствам структурного программирования могли бы оказаться удобными и в таких языках, как Алгол, ПЛ/1 и Кобол.

Один случай, заслуживающий специального упоминания, касается ситуации «конец файла». Программист, работающий на Коболе, привык, например, к следующей записи:

```
READ FILEX AT END GO TO END-ROUTINE.
```

Аналогично программирующий на Фортране знает, что в записи оператора ввода-вывода он должен указать номера операторов для ситуаций «конец файла» и «ошибка четности»; программирующий на ПЛ/1 пользуется в этой и многих других ситуациях конструкцией ON CONDITIONS.

Ситуация «конец файла» и некоторые другие в действительности являются примерами *прерываний,* т. е. ситуаций, в которых мы допускаем возможность нарушения нормального хода процесса обработки. В структурированном подходе применению в этой ситуации оператора GO TO можно было бы предпочесть среди прочих средств введение логического признака, значение которого в последующем проверяется. Так, на Коболе программист мог бы записать

```
READ FILEX AT END MOVE 1 TO EOF-FLAG.
IF EOF-FLAG=1 THEN PERFORM ENDROUTINE
ELSE
   MOVE A TO B
   MOVE C TO D


   и т. д
```

Аналогичные соглашения могут быть легко установлены в **ПЛ/1** и других языках программирования. В какой мере это загромождает программный код, является вопросом спорным; главная идея состоит в том, что в большинстве ситуаций *может* быть найдено решение, согласующееся с основными принципами структурного программирования.

4.5.4. Отвечают ли современные алгоритмические языки требованиям структурного программирования?

В разд. 4.3.4 мы уже обсудили реализацию структурного программирования на таких языках, как Алгол, ПЛ/1, Кобол, Фортран и языки ассемблера. Вряд ли остаются какие-нибудь сомнения в том, что Алгол и ПЛ/1 являются исключительными средствами реализации структурного программирования. На самом деле сторонники этих языков питают некоторую надежду, что интерес к структурному программированию ослабит тягу программистов к Фортрану и Коболу и привлечет их внимание к Алголу и ПЛ/1. Остается несколько вопросов, касающихся эффективности, но они не кажутся нам настолько серьезными, чтобы лишить желания пользоваться этими языками.

По мнениюмногих сторонников структурного программирования, Коболу недостает формализованной структуры, которая желательна при построении программ без использования операторов GO TO. Как мы видели выше, отсутствие явной блочной структуры, несколько неоднозначная форма вложенного оператора IF и громоздкость оператора PERFORM-UNTIL могут иногда быть причиной неудобств в структурном программировании. Тем не менее *стандартный* язык Кобол *может* быть использован для написания структурированных программ, как в настоящее время и поступают в некоторых организациях. Тщательный анализ опыта его использования в программных разработках должен показать, компенсируются ли неэффективность и неудобства структурного программирования на Коболе таким выигрышем, как более высокая производительность программистов и более низкая стоимость тестирования.

Между тем некоторые группы программирующих на Коболе начали обнаруживать, что их версия Кобола (т. е. реализация языка Кобол, поставленная им с ЭЗМ) не обладает некоторыми свойствами, обеспечивающими удобство структурного программирования. Некоторые, например, установили, что в их версии Кобола вложенные операторы IF компилируются с ошибками (т. е. генерируется ошибочный объектный код); другие нашли, что в их компиляторе ограничивается число используемых в программе операторов PERFORM; кто-тоещеустановил, чтових версии Кобола процедура входа в подпрограмму SORT должна быть организована в виде СЕКЦИИ, из которой нельзя выходить по оператору PERFORM. Было даже заявлено, что в одной версии Кобола (на машине, которую мы не будем называть) не содержится оператор GO TO DEPENDING ON; другие установили, что оператор GO TO DEPENDING ON генерирует объектный код, аналогичный объектному коду, соответствующему вложенному оператору IF-THEN-ELSE.

Фортран в нынешнем его виде *не* представляется хорошим средством структурного программирования. Если движение против «go to» найдет всеобщее признание, то весьма вероятно, что мы увидим некоторые дополнения к Фортрану (каждое из которых могло бы обрабатываться некоторым препроцессором, вырабатывающим в качестве выхода исходный код на *стандартном* Фортране и при этом вынуждающим программиста пользоваться структурными методами). Кстати сказать, Фортран, как представляется, используется главным образом научными работниками, инженерами и математиками, которые не слишком интересуются теми выгодами, которые представляет структурное программирование, хотя имеются и обратные примеры.

Языки ассемблеров, макроязыки системных разработок могли бы служить хорошим аппаратом структурного программирования, и кажется большинство современных работ в этом направлении ведется на этих языках. Названные языки используются главным об-

разом в *системном программировании,* где забота об отсутствии ошибок и простоте тестирования является естественной. То обстоятельство, что эти языки низкого уровня позволяют программисту более непосредственно управлять устройствами машины, снимает многие касающиеся эффективности вопросы, за исключением тех немногих, которые внутренне свойственны подходу структурного программирования (например, возникающая иногда необходимость дублирования некоторых разделов программного кода).

ЛИТЕРАТУРА

1. Dijkstra E. W.,ProgrammingConsidered as a Human Activity, Proceedings of IFIP Congress 65, Spartan Books, Washington, D. C., 1965.
2. Dijkstra E. W., Go-To Statement Considered Harmful, Letter to the Editor, Communications of the ACM, March 1968.
3. Dijkstra E. W., The Structure of the THE-Multiprogramming System, Communications of the ACM, May 1968, p. 3-141—3-146 Copyright 1968, Association for Computing Machinery, Inc., reprinted by permission.
4. Dijkstra E. W., Structured Programming, Software Engineering Techniques, NATO Scientific Affairs Division, Brussels 39, Belgium, p. 84—88.
5. Dijkstra E. W., Notes on Structured Programming, EWD 249, Technical University, Eindhoven, Netherlands, 1969; имеется русский перевод: Дейкстра Э., Заметки по структурному программированию, в книге Дал У., Дейкстра Э., Хоор К., Структурное программирование, изд-во «Мир», М., 1975.
6. Van Wijndgaarden A., Recursive Definition of Syntax and Semantics, Formal Language Description Languages for Computer Programming, ed. T. B. Steel, Jr., North-Holland, Amsterdam, 1966.
7. Van Schorre D., Improved Organization for Procedural Languages, Technical Memo, System Development Corp., Santa Monica, California, Aug. 1966.
8. Landin P. J., The Next 700 Programming Languages, Communications of the ACM, March 1966, p. 157—164.
9. Aron J. D., The Superprogrammer Project, Software Engineering Techniques, NATO Scientific Affairs Division, Brussels 39, Belgium, p. 50—52.
10. Baker F. T., Chief Programmer Team Management of Production Programming, IBM Systems Journal, Jan. 1972, p. 56—73.
11. Wulf W. A., A Case Against the GOTO, Proceedings of the 25th ACM National Conference, v. 2, 1972, p. 791—797. [Copyright 1972, Association for Computing Machinery, Inc., reprinted by permission.]
12. Hopkins M. E., A Case for the GOTO, Proceedings of the 25th ACM National Conference, v. 2, 1972, p. 787—790. [Copyright 1972, Association forComputing Machinery, Inc., reprinted by permission.]
13. Software Enginerring, eds. P. Naur, B. Randell, NATO Scientific Affairs Division, Brussels 39, Belgium, published in January 1969.
14. Software Engineering Techniques, eds. J. N. Buxton, B. Randell, NATO Scientific Affairs Division, Brussels 39, Belgium, published April 1970.
15. King J., A Program Verifier, Ph. D. Thesis, Carnegie-Mellon University, 1969.
16. Manna *Z.,* Ness S., Vaillemin J., Inductive Methods forProperties About Programs, SIGPLAN/SIGACT Conference on Proving Assertions About Programs, Jan. 1972.
17. Hoare C. A. R., Proof of a Program: FIND, Communications of the ACM, Jan. 1971, p. 39—45.
18. Aron J. D., Estimating Resources for Large Programming Systems, Software Engineering Techniques, NATO Scientific Affairs Division, Apr. 1970, p. 68—79.
19. Kleene S. C., Introduction to Metamathematics, D. Van Nostrand Co., Inc., New York, 1952; имеется русский перевод: Клини С. К., Введение в метаматематику, ИЛ, 1957.

20. Post E. I., Finite Combinatory Processes-Formulation I, Journal of Symbolic Logic, v. 1, 1936.
21. Марков А. А., Теория алгоритмов, Труды Математического ин-та АН СССР им. В. А. Стеклова, изд-во АН СССР, 1954.
22. Church A., The Calculi of Lambda-Conversion, Annalsof Mathematical Studies, №6, Princeton University Press, Princeton, New Jersey, 1951.
23. Leavenworth B. M., Programming Without the GOTO, Proceedings of the 25th ACM National Conference, v. 2, p. 782—786.
24. Böhm C., Jacopini G., Flow Diagrams, Turing Machines, and Languages with Only Two Formulation Rules, Communicatins of the ACM, May 1966.
25. Knuth D. E., Floyd R. W., Notes on Avoiding GOTO Statements, Technical Report CS148, Stanford University, Jan. 1970,
26. Aschcroft E., Manna Z., The Translation of «Goto» Programs into «While»Programs, Proceedings of 1971 IFIP Congress.
27. Manugian M. G., A Collection of Readings on the Subject of BLISS-10, DECUS Program Library, DECUS № 10-118, Part II, Digital Equipment Corporation, Maynard, Massachusetts, Dec. 1971.
28. Wulf W. A., Programming Without the GOTO in A Collection of Readings on the Subject of BLISS-10, p. 3-1—3-25.
29. Wulf W. A. et al., BLISS a Language for Systems Programming, Communications of the ACM, Dec. 1971.
30. Wirth N.. PL360, A Programming Language for the 360 Computers, Journal of the ACM, Jan. 1968, p. 37—74.
31. Conrow K., Smith R. G., NEATER2: a PL/I Source Statement Reformatter, Communications of the ACM, Nov. 1970.
32. Schmalz R. J., OS/360 PL/I-F Performance Guidelines, IBM, Dept. H74, Poughkeepsie. New York, June 1969.
33. McCracken D., Revolution in Programming: An Overview, Datamation, Dec. 1973, p. 50—51.
34. Donaldson J. R., Structured Programming, Datamation, Dec. 1973, p. 52—54.
35. Miller E. F., Lindamood G. E., Structured Programming: Top-Down Approach, Datamation, Dec. 1973, p. 55—57.
36. Baker F. T., Mills H. D., Chief Programmer Teams, Datamation, Dec. 1973, p. 58—61.
37. Clark R. L., A Linguistic Contribution to GOTO-Less Programming, Dec. 1973, p. 62—68.
38. Weinberg G., Structured Programming in PL/C, Wiley, Inc., 1972.
39. Dahl O. J., Dÿkstra E. W., HoareC. A. R., Structured Programming, Academic Press, 1972, имеется русский перевод: Дал У., Дейкстра Э., Хоор К., Структурное программирование, изд-во «Мир», М., 1975.
40. Flynn J., SEFTRAN User Guide, Interoffice Computing Memorandum № 337, Jet Propulsion Laboratory, 4800 Oak Grove Drive, Pasadena, California 91103.
41. Lee R. C. T., Chang S. K., Structured Programming and Automatic Program-Synthesis, Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages, March 1974.
42. Karpinski R. H., An Unstructured View of Structured Programming, ACM SIGPLAN Notices, v. 9, №3, March 1974.
43. Baker F. T., System Quality Through Structured Programming, Proceeding 1972 Fail Joint Computer Conference, p. 339—342.
44. Mills H., Mathematical Foundations for Structured Programming—Report FSC72-6012, IBM Corporation, Gaithersburg, Md., 1972.
45. Parnas D., On the Problem of Producing Well Structured Programs, Computer Science Research Review, Carnegie—Mellon University, Pitsburgh, Pa., 1972.
46. McGowan C.L., Kelly J. R., Top-Down Structured Programming, Mason and Lipscomb, 1975.
47. Yourdon E., A Brief Look at Structured Programming and Top-Down Design, Modem Data, June 1974, p. 30—35.

48. Wirth N., On the Composition of Well-Structured Programs, ACM Computing Surveys, Dec. 1974, p. 247—260.
49. Knuth D. E., Structured Programming with Go-To Statements, ACM Computing Suiveys, Dec. 1974, p. 261—302.

ВОПРОСЫ

1. Дайте определение структурного программирования одним-двумя абзацами. Какое отношение между структурным программированием и оператором GO TO обычно наблюдается в языках программирования высокого уровня? Будьте точны: не прибегайте к той слишком упрощенной формулировке, что «структурное программирование — программирование без операторов GO TO».

2. Почему, на ваш взгляд, структурное программирование не было общепринятым в течение столь долгого времени? Считаете ли вы, что в вашей организации противодействие внедрению структурного программирования будет не столь упорным?

3. В чем состоят главные цели структурного программирования? Дайте их краткое описание.

4. Почему структурированные программы должны быть проще в отладке? Какого характера тестирование обычно применяется к структурированным программам?

5. Почему структурное программирование позволяет программисту писать большее число отлаженных команд в день? Чго можно ожидать в связи с этим для типичной программной разработки?

6. Почему структурированные программы отличаются от типичных неструктурированных программ лучшей читабельностью? Считаете ли вы, что это существенный довод в пользу структурного программирования?

7. В чем разница между структурным программированием и модульным программированием?

8. Назовите основные составляющие блоки, предложенные Бомом и Джакопипи.

9. В каком отношении находятся структурное программирование и методология нисходящего проектирования, рассмотренная в гл. 2?

10. Перечислите основные операторы и конструкции используемого вами языка программирования, необходимыедля написания структурированных программ.

11. При каких условиях можно пользоваться оператором GO TO, не нарушая основных принципов структурного программирования?

12. Рассмотрите одно из расширений основных составляющих блоков структурного программирования и покажите, что в нем не нарушаются принципы этого подхода.

13. Некоторые программисты высказывали мнение, что в расширенном варианте основных правил структурного программирования следует разрешить множественные выходы (например, «нормальный» выход и выход «по ошибке»). Как это может быть сделано в рамках требований структурного программирования (т. е. по принципу «один вход — один выход»)?

14. Некоторые программисты высказывали мнение, что следовало бы разрешить пользоваться операторами GO TO для передач управления *вперед* по ходу программы. В каком случае это могло бы привести к программе, которую трудно понимать и отладить?

15. На рис. 4.19 показана блок-схема неструктурированной программы. Нарисуйте блок-схему эквивалентной структурированной программы, пользуясь методом дублирования кодов. Нанесением соответствующих границ выделите «черные ящики», составляющие структурированную версию блок-схемы.

16. На рис. 4.20 показана блок-схема неструктурированной программы. Нарисуйте блок-схему эквивалентной структурированной программы, пользуясь методом дублирования кодов. Нанесением соответствующих границ выделите «черные ящики», составляющие структурированную версию блок-схемы.

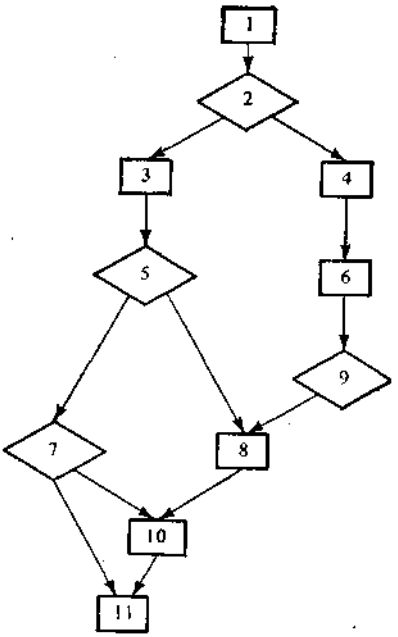


Рис. 4.19.

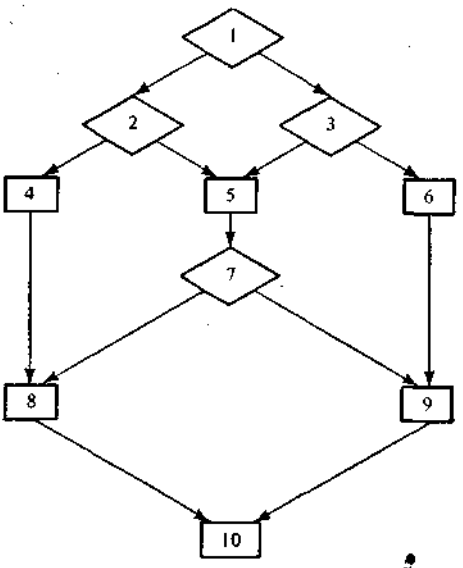


Рис. 4.20.

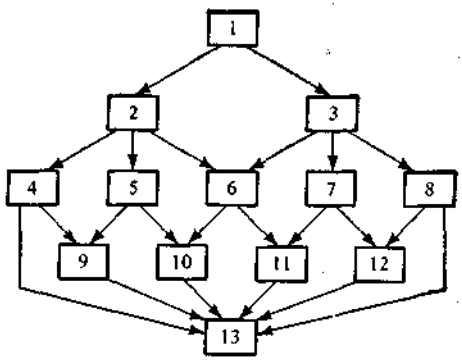


Рис. 4.21.

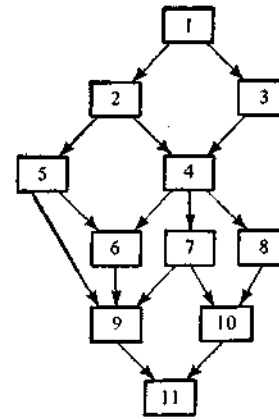


Рис. 4.22.

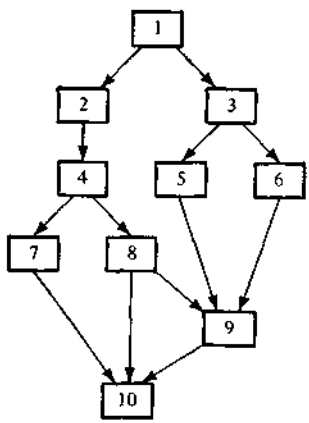


Рис. 4.23.

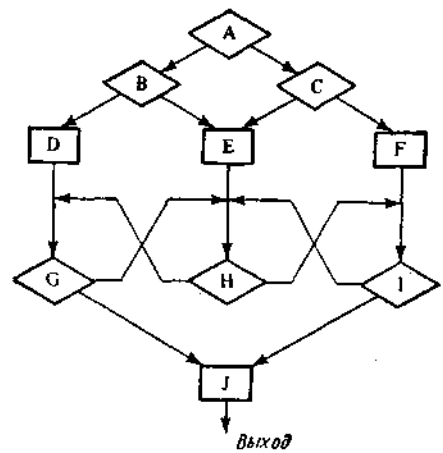


Рис. 4.24.

17. На рис. 4.21 показана блок-схема неструктурированной программы. Нарисуйте блок-схему эквивалентной структурированной программы, пользуясь методом дублирования кодов. Нанесением соответствующих границ выделите «черные ящики», составляющие структурированную версию блок-схемы.

18. На рис. 4.22 показана блок-схема неструктурированной программы. Нарисуйте блок-схему эквивалентной структурированной программы, пользуясь методом дублирования кодов. Нанесением соответствующих границ выделите «черные ящики», составляющие структурированную версию блок-схемы.

19. На рис. 4.23 показана блок-схема неструктурированной программы. Нарисуйте блок-схему эквивалентной структурированной программы, пользуясь методом дублирования кодов. Нанесением соответствующих границ выделите «черные ящики», составляющие структурированную версию блок-схемы.

20. На рис. 4.24 показана блок-схема неструктурированной программы. Нарисуйте блок-схему эквивалентной структурированной программы, пользуясь методом Ашкрофта — Манны.

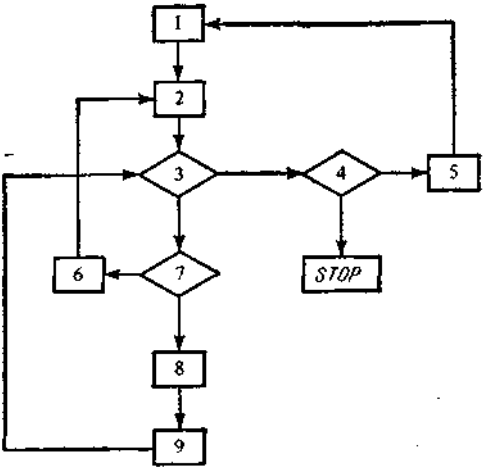


Рис. 4.25.

21. На рис. 4.25 показана блок-схема неструктурированной программы. Нарисуйте блок-схему эквивалентной структурированной программы, пользуясь методом Ашкрофта — Манны.

22. На рис. 4.26 показана блок-схема неструктурированной программы. Нарисуйте блок-схему эквивалентной структурированной программы, пользуясь методом Ашкрофта — Манны.

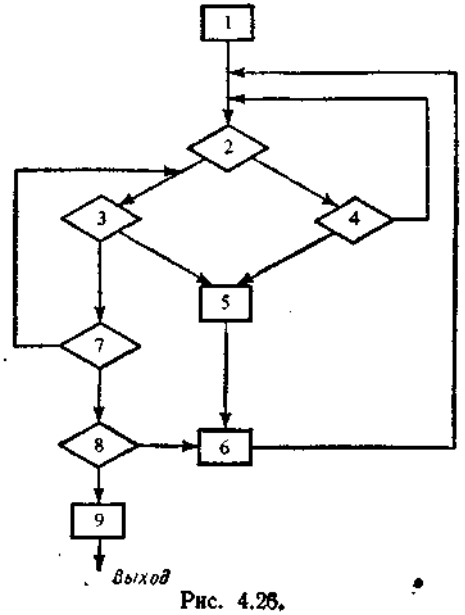


Рис. 4.26.

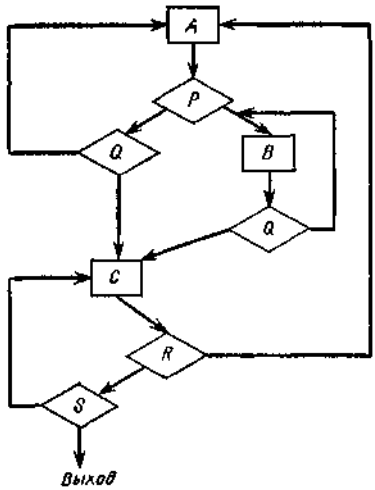


Рис. 4.27.

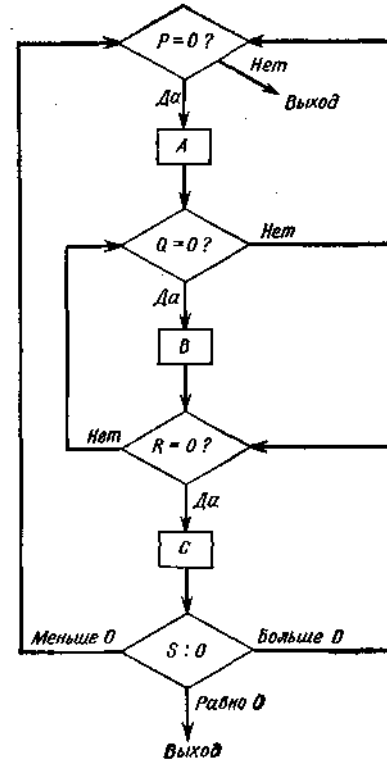


Рис. 4.28.

23. На рис. 4.27 показана блок-схема неструктурированной программы. Нарисуйте блок-схему эквивалентной структурированной программы, пользуясь методом Ашкрофта — Манны.

24. На рис. 4.28 показана блок-схема неструктурированной программы. Нарисуйте блок-схему эквивалентной структурированной программы, пользуясь методом Ашкрофта — Манны.

25. На рис. 4.29 показана маленькая программа, написанная в неструктурированном виде. Перепишите ее, пользуясь методом булева признака. Неструк-

```
          Присвоить переменной I значение 1,
          Присвоить переменной FLAG значение 0.
ЦИКЛ.     Если TABLE (I) > N, то перейти к ХОРОШО.
          Если TABLE (l) = 0, то перейти к ПЛОХО.
          Присвоить переменной I значение TABLE(I + 1)
          Перейти к ЦИКЛ.
ХОРОШО.   Присвоить переменной FLAG значение 1.
          Перейти к ГОТОВО.'
ПЛОХО.    Присвоить переменной FLAQ значение 0.
ГОТОВО.   Выход.
```

Рис. 4.29.

```
ЦИКЛ1.   Читать с диска запись в FLAG.
         Если FLAG  0, то перейти к ЦИКЛ1.
         Писагь А на диск.
         Читать с лиска запись в FLAG.
         Если FLAG  А, то перейти к ЦИКЛ1.
         Выход.
```

Рис. 4.30.

```
ЦИКЛ.    Присвоить переменной I значение (START + FINISH)/2.
         Если TABLE(I) = ITEM, то перейти к НАЙДЕНО.
         Если TABLE(I) < ITEM, то присвоить START значение (I + 1).
         Если TABLE(I) > ITEM, то присвоить FINISH значение (I — 1).
         Если (FINISH — START) > 1, то перейти к ЦИКЛ.
         Если TABLE(START) = ITEM, то перейти к НАЙДЕНО.
         Если TABLE(FINISH) = ITEM, то перейти к НАЙДЕНО,
НАЙДЕНО. Присвоить переменной FLAG значение 0.
ГОТОВО.  Присвоить переменной FLAG значение 1,
         Выход.
```

Рис. 4.31.

```
         Присвоить переменной I значение 1,
ЦИКЛ.    Присвоить переменной J значение 1.
СРАВНЕНИЕ, Если TABLE(J) < TABLE(J + 1), то перейти к ПРИРАЩЕНИЕ.
         Присвоить переменной TEMP значение TABLE(J).
         Присвоить переменной TABLE(J) значение TABLE(J+1),
         Присвоить переменной TABLE(J + 1) значение TEMP,
ПРИРАЩЕНИЕ. Присвоить переменной J значение J + 1 .
         Если J < N, то перейти к СРАВНЕНИЕ,
         Присвоить переменной I значение I + 1,
         Если I < (N+1), то перейти к ЦИКЛ."
         Выход.
```

Рис. 4.32.

вурированная версия этой программы написана на «машинном эсперанто», применение которого вам должно быть совершенно ясным. Составляя структурированную версию программы, воспользуйтесь одним из конкретных языков, например Коболом, Фортраном, Алголом и т. д.

26. На рис. 4.30 показана маленькая программа, написанная в неструктурированном виде. Перепишите ее, пользуясь методом булева признака. Неструктурированная версия этой программы написана на «машинном эсперанто», применение которого вам должно быть совершенно ясным. Составляя структурированную версию программы, воспользуйтесь одним из конкретных языков, например Коболом, Фортраном, Алголом и т. д.

27. На рис. 4.31 показана маленькая программа, написанная в неструктурированном виде. Перепишите ее, пользуясь методом булева признака. Неструктурированная версия этой программы написана на «машинном эсперанто», применение которого вам должно быть совершенно ясным. Составляя структурированную версию программы, воспользуйтесь одним из конкретных языков, например Коболом, Фортраном, Алголом и т. д.

28. На рис. 4.32 показана маленькая программа.написаннаявнеструктурированном виде. Перепишите ее, пользуясь методом булева признака. Неструктурированная версия этой программы написана на «машинном эсперанто», применение которого вам должно быть совершенно ясным. Составляя структурированную версию программы, воспользуйтесь одним из конкретных языков, например Коболом, Фортраном, Алголом и т. д.

```
                Присвоить переменной MAX значение N.
                Присвоить переменной I значение 1.
ЦИКЛ.           Присвоить переменной J значение 1.
                Присвоить переменной FLAG значение 0.
СРАВНЕНИЕ.      Если TABLE(J) < TABLE(J+ 1), то перейти к ПРИРАЩЕ-
                НИЕ.
                Присвоить переменной TEMP значение TABLE(J).
                Присвоить переменной TABLE(J) значение TABLE(J+1).
                Присвоить переменной TABLE(J+1) значение TEMP.
                Присвоить переменной FLAG значение 1.
ПРИРАЩЕНИЕ.     Присвоить переменной J значение J + 1 .
                Если J < MAX, то перейти к СРАВНЕНИЕ.
                Если FLAG = 0, то перейти к ГОТОВО.
                Присвоить переменной I значение (I+1).
                Присвоить переменной MAX значение (MAX — 1).
                Если I < (N+1), то перейти к ЦИКЛ.
ГОТОВО, •       Выход.
```

Рис. 4.33.

29. На рис. 4.33 показана программа, написанная в неструктурированном виде. Перепишите эту программу, пользуясь методом булева признака. Неструктурированная версия этой программы написана на «машинном эсперанто», применение которого вам должно быть совершенно ясным. Составляя структурированную версию программы, воспользуйтесь одним из конкретных языков, например Коболом, Фортраном, Алголом и т. д.

30. В чем состоят недостатки метода преобразования дублированием кодов? Существенны ли эти недостатки?

31. Почему логические ошибки менее вероятны в программе, составленной с применением метода дублирования кодов, чем в эквивалгнтной неструктурированной программе?

32. В чем состоятосновныевозражения против преобразования неструктурированной схемы в структурированную по методу Ашкрофта — Манны? Существенны ли эти возражения?

33. Блок-схема, показанная на рис. 4.17а, представляется излишне упрощенной версией реального алгоритма обновления главного файла. Напишите *структурированную* версию более полной программы обновления, в которой

а) первое поле записи содержит ключ первичной идентификации.

б) главный файл и файл сообщений упорядочиваются по этому ключу (в возрастающей последовательности).

в) второе поле в записях сообщений занято кодом функции, указывающим, следует ли внести эту запись в главный файл или использовать информацию о записи при изменении различных полей соответствующей записи в главном файле, или стереть соответствующую запись главного файла.

г) одной и той же записи главного файла могут отвечать несколько записей файла сообщений (расположенных в этом файле одна за другой).

34. Почему метод Ашкрофта — Манны упрощает отладку?

35. Было высказано предположение, что неструктурированную программу, показанную на рис. 4.34а, можно было бы переписать в структурированной форме, представленной рис. 4.346. Найдите какой-нибудь *другой* способ написания этой программы, применяя метод булева признака *и* не вводя каких-либо признаков в программу.

36. Было высказано предположение, что неструктурированную программу, показанную на рис. 4.35а, можно переписать в структурированной форме, представленной рис. 4.356. Найдите какой-нибудь *другой* способ написания этой программы, применяя метод булева признака и не вводя какнх-либо новых признаков в программу.

```
          MOVE 1 TO I.
          MOVE 0 TO FLAG.
LOOP.     IF TABLE(I) > N GO TO GOOD,
          IF TABLE(I) = 0 GO TO BAD.
          MOVE TABLE(I + 1) TO 1,
          GO TO LOOP.
GOOD.     MQVE 1 TO FLAG.
          GO TO DONE.
BAD.      MOVE 0 TO FLAG,
DONE,     EXIT.
```

Рис. 4.34а.

```
          MOVE 1 TO I.
          MOVE 0 TO FLAG, DONE.
          PERFORM LOOP UNTIL DONE = 1,
          EXIT.
LOOP.     IF TABLE(I) > N
            MOVE 1 TO FLAG, DONE
          ELSE
            IF TABLE(1) = O
               MOVE I TO DONE
               MOVE 0 TO FLAG
            ELSE MOVE TABLE(I+I) TO I.
          EXIT.
```

Рис. 4.346.

```
LOOP1,    PERFORM READ-DISK-INTO-FLAG.
          IF FLAG = O GO TO LOOP1.
          PERFORM WRITE-A-ONTO-DISK.
          PERFORM READ-DISK-INTO-FLAG
          IF FLAG   A GO TO LOOP1.
          EXIT.
```

Рис. 4.35а.

```
          MOVE 9999 TO FLAG.
          PERFORM OUTER-LOOP UNTIL FLAG = A,
          EXIT.
OUTER-LQOP,  PERFORM INNER-LOOP UNTIL FLAG = 0,
          PERFORM WRITE-A-ONTO-DISK.
          PERFORM READ-DISK-INTO-FLAG,
          EXIT.
INNER-LOOP.  PERFORM READ-DISK-INTO-FLAG,
          EXIT.
```

Рис. 4.356.

37. Какие свойства Кобола делают структурное программирование на нем затруднительным?

38. Какие свойства Фортрана делают структурное программирование на нем затруднительным и неудобным? При этих ограничениях, как можетиспользоваться Фортран для написания структурированных программ?

39. Исследуйте компилятор Кобола вашей машины и установите, нет ли в нем каких-нибудь специальных ограничений, которые делали бы структурное программирование на нем затруднительным (например, ограничение на число обращений по оператору PERFORM).

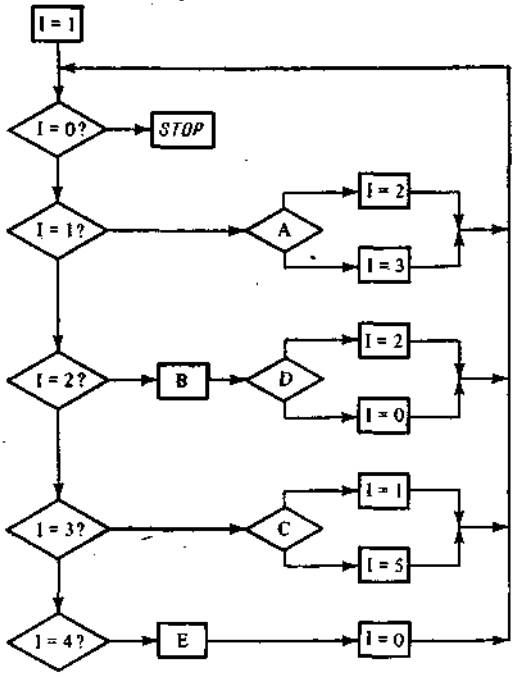


Рис. 4.36. Упрощенный вариант схемы по рис. 4.15.

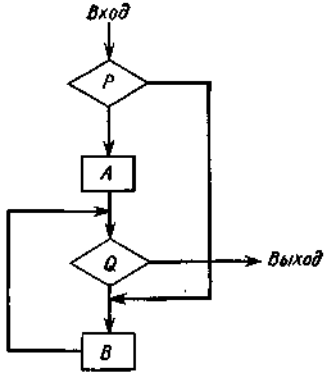


Рис. 4.37. Неструктурированный цикл.

40. (Эксперимент.) Выберите нетривиальную программу из числа тех, с которыми вы обычно работаете в своей организации; подберите приятеля или коллегу приблизительно равных с вашими способностями в программировании. Один из вас пусть составит программу в соответствии с принципами структурного программирования, другой — программу, отличающуюся максимально возможной эффективностью (что, быть может, потребует некоторых нарушений этих принципов). По завершении работы сравните программы и установите, много ли дополнительного времени ЦП и памяти требует реализация принципов структурного программирования. Значительны ли эти потери? Считаете ли вы, что в программировании других задач издержки характеризовались бы такими же значениями?

41. На рис. 4.13 показан образец небольшой блок-схемы; рис. 4.15 изображает блок-схему той же программы после преобразования по методу Ашкрофта — Манны. Заметьте, что в схеме рис. 4.15 переменной состояния / присваивается значение 4 только в одном месте. Это означает, что мы могли бы упростить схему 4.15, переписав ее в форме, представленной рис. 4.36. Продолжите этот процесс, т. е. упростите схему рис. 4.36, выделяя те состояния, которые упоминаются только однажды.

42. Пользуясь приемом, описанным в п. 41, упроститерешение по методу Ашкрофта — Манны задачи по рис. 4.24 (см. также п. 20).

43. Пользуясь приемом, описанным в п. 41, упростите решение по методу Ашкрофта — Манны задачи по рис. 4.25 (см. также п. 21).

44. Пользуясь приемом, описанным в п. 41, упростите решение по методу Ашкрофта — Манны задачи по рис. 4.26 (см.,также п. 22).

45. Пользуясь приемом, описанным в п. 41, упростите решение по методу Ашкрофта — Манны задачи по рис. 4.27 (см. также п. 23).

46. Пользуясь приемом, описанным в п. 41, упростите решение по методу Ашкрофта — Манны задачи по рис. 4.28 (см. также п. 24).

47. На рис. 4.16а показана обычная форма неструктурированного цикла, т. е. цикла с одним входом и двумя выходами. Рис. 4.16а показывает, каким образом можно структурировать такой цикл введением в программу нового признака; альтернативный способ структурирования этой программы без введения нового признака показан на рис. 4.16в. Рассмотрим теперь неструктурированный цикл, показанный на рис. 4.37; обратите внимание на то, что в этом цикле два входа й один выход. Перестройте цикл таким образом, чтобы он обладал соответствующей структурой. Пример, иллюстрируемый рис. 4.16а, 4.16б, 4.16в, должен подсказать вам" правильный путь решения.

ГЛАВА 5

СТИЛЬ В ПРОГРАММИРОВАНИИ: ПРОСТОТА И ЯСНОСТЬ[1)]

...насколько простым становится процесс проектирования, если мы пользуемся критерием: «это должно легко объясняться». Если делается попытка ввести в систему нечто, требующее для своего объяснения более строки или абзаца, откажитесь от нее — она бесполезна.

Смит
Software Engineering, p, 71.

Я думаю нам следует сравнить наши методы управления с порядком подготовки технических чертежей. В больших организациях чертеж обычно подписывает его исполнитель и непосредственный руководитель, подтверждающий, что чертеж отвечает соответствующим нормам. В программировании вы обычно не встречаетесь с этой второй подписью — по правде сказать, не бывает и первой. Ясности изложения и стилю не придают, по-видимому, никакого значения — лишь бы программа работала должным образом. Мне представляется важным, чтобы программы удовлетворяли еще и некоторым эстетическим нормам.

Макилрой
Software Engineering, p. 89.

Д-р Макклюр: я знаю, что в очень редких программирующих организациях непосредственные руководители дают себе труд прочесть программный код, написанный их подчиненными, и попытаться понять его. Я полагаю, что это совершенно необходимо.
Проф. Бакстон: я знаю, что в очень редких программирующих организациях руководитель способен прочесть программный код; к присутствующим это, естественно, не относится.

Software Engineering, p. 89.

5.0. Введение

Прямо или косвенно в нескольких последних главах мы старались подчеркнуть, что каждый программист *часто склонен переоценивать свои способности*. Мы показали, например, что нисходящая схема, предложенная в гл. 2, обычно является наиболее логичной и организованной формой проектирования программ. Тем не менее немно-

1) См. примечание на стр. 13.

гие программисты на самом деле пользуются этим подходом, потому что часто они полагают, что могут спроектировать законченную программу «сходу»; кажется, что они рассчитывают (по крайней мере, они поступают, как если бы они так рассчитывали) на то, что они могут спроектировать свою программу в *процессе ее кодирования.*

Предметом рассмотрения предыдущих глав было следующее положение: *программирование требует внимательного, осторожного и организованного подхода.* Довольно трудно добиться того, чтобы большая программа выполнялась правильно; не следует вводить дополнительные трудности, обращаясь без особой нужды к изощренным и сложным алгоритмам. Аналогичный принцип принят во многих технических отраслях; он часто упоминается как принцип KISS[1]. Повторяем, что это всего лишь *принцип;* мы не располагаем теоремами или статистическими исследованиями, которые могли бы служить доказательством того, что простота проекта обеспечивает лучшую программу. Тем не менее обычно можно наблюдать, что лучшие программисты сознательно или подсознательно пользуются этим принципом.

Предыдущие главы неоднократно касались и другого предмета: в конечном итоге большинство программ используется и модифицируется другими программистами. Одним неофициальным исследованием несколько лет назад было установлено, что в среднем американский программист меняет место работы каждые 1,4 года. Хотя и нет соответствующей статистики, но мы могли бы оценить среднее время жизни программы для ЭВМ, увеличив этот срок вдвое, если не больше[2]. Отсюда следует совершенно определенный вывод: *если имеется в виду, что рано или поздно программы должны сопровождаться кем-то другим, то их следует писать так, чтобы их можно было прочесть и понять.* Как отметил Дж. Вейнберг в своей замечательной книге The Psychology of Computer Programming (Психология программирования), возникающие здесь трудности частично связаны с тем, что многие программисты считают составленные ими программы личной собственностью.

Настоящая глава имеет целью подробно рассмотреть две названные выше проблемы. Во-первых, что может быть сделано для упрощения программирования, дабы на этапах кодирования и отладки мы сталкивались с меньшими трудностями? Во-вторых, каким образом добиться того, чтобы облегчить чтение и понимание программ тем, кто их не составляет? Мы начнем с краткого обзора некоторых рекомендаций предшествующих глав, многие из которых рассматрива-

[1] KISS — Keep It Simple Stupid («Будь попроще, дурачок».— *англ.).— Прим. перев.*

[2] В одном неофициальном обзоре крупной организации, разрабатывающей ЭВМ, отмечалось, что, после того как программа кем-то написана, обычно, прежде чем ее переделают заново, она будет сопровождаться программистами следующих десяти поколений.

лись как средство упрощения программ (с тем чтобы ускорить их отладку, тестирование и т. д.). Затем мы обсудим ряд методов и приемов программирования, способствующих повышению читаемости разрабатываемых программ.

5.1. Обзор предложений по разработке простых программ

Многие из предложений, сформулированных в гл. 1, 3 и 4, имели целью упростить отладку и тестирование программ, а также их сопровождение и модификацию. Само собой разумеется, что большая часть этих предложений может быть использована для того, чтобы сделать программы более понятными. В частности, предложения, рассматриваемые ниже, рекомендуются как средства достижения простоты.

5.1.1. Пользуйтесь методами структурного программирования

В гл. 4 рассмотрен подход к разработке программ, известный как «структурное программирование». Одной из причин возникновения этого подхода было желание иметь более «читабельные» программы и программные листинги. Исключая программы с сетевыми структурами, которые часто возникают при неограниченном употреблении операторов GO TO, мы значительно снижаем возможности программиставнеупорядоченной организации листингов его программ.

5.1.2. Без крайней необходимости избегайте многоцелевого применения

Свойственные многим современным операционным системам возможности реализации многоцелевого применения являются чрезвычайно мощными средствами. Они удобны в приложениях, связанных с совмещением вычислений и операций ввода-вывода, систем коллективного использования и систем, работающих в реальном масштабе времени, атакже в других случаях, где используются операции, несогласованные во времени. Однако структура большинства программ многоцелевого применения очень сложна; существует ряд тонких ошибок, которые могут долго сохраняться в такой программе, после того как вы сочли ее уже отлаженной, а потом внезапно обрушиться на какого-то несчастливца, мучительно пытающегося понять, что же в такой программе происходит. Поэтому без крайней необходимости вам следует стараться избегать использования таких операторов, как ATTACH и DELETE в Системе IBM/360, или оператор FORK в языке XSS-940, или оператор ZIP в языке B5500 фирмы Burroughs.

5.1.3. Не допускайте произвольного употребления системы команд или языка программирования

Как мы отметили в гл. 1, в некоторых машинах встречаются *неопределенные* команды; их исполнение иногда приводит к довольно странным, неописанным в официальных руководствах, но тем не менее предсказуемым результатам. Во многих случаях, однако, результат оказывается совершенно непредсказуемым. К счастью, в большинстве современных машин неопределенные коды операций встречаются довольно редко; тем не менее некоторые программисты, работающие на языке ассемблера, все-таки находят способы *непредписанного использования* системы команд их машины. Стоит еще раз повторить вывод, сформулированный в гл. 1: не пытайтесь приспособить систему команд к вашей специальной задаче, если у вас нет полной уверенности в своих действиях; если вы понимаете, что вам необходимо отступить от исходного назначения данной команды, то вам следует позаботиться о ясном и достаточно полном комментарии ваших действий в листинге программы, чтобы они были понятны тем, кто будет сопровождать программу после вас.

Аналогичные замечания уместны, конечно, и в отношении языков программирования высокого уровня. Программирование на Фортране, кажется, особенно искушает к применению подобных приемов; в Коболе программисты нашли ряд приемов для выполнения таких функций, как присваивание в качестве начального значения таблицы некоторой константы; большинство этих приемов страдает отсутствием ясности и к тому же они связаны с определенной машиной. В большинстве случаев лучше их избегать и пользоваться только стандартными средствами языка.

5.1.4. Не пишите программы, которые модифицируются в процессе исполнения

Почти все языки программирования позволяют программисту модифицировать программу во время ее исполнения. В программе на языке ассемблера такие действия выступают в явной форме; в языках высокого уровня модификация программы иногда маскируется такими эффектными терминами, как «переключатели программ» и т. д. Так, оператор ALTER в Коболе фактически позволяет программисту изменять программный код во время исполнения, назначенный оператор GO TO в Фортране в определенной степени характеризуется тем же свойством, возможность в ПЛ/1 определить переменную типа метки может рассматриваться как более формализованная и утонченная форма назначенного GO TO.

Среди недостатков модифицируемых во время исполнения программ можно выделить два, которые представляются не очень широко известными: такие программы обычно *нереентерабельны* и

возможность их использования в *последовательности многократных обращений* менее вероятна. Реентерабельной называется программа, которая может конкурентно (а в мультипроцессорных системах одновременно) использоваться двумя или большим числом процессов. Ясно, что если программа изменяется во время ее исполнения, то могут возникнуть серьезные трудности при конкурентном ее использовании двумя или более заданиями или пользователями. Теоретически назначенный оператор GO TO в Фортране и операторы, использующие переменные типа метки, в языке ПЛ/1 не обязательно приводят к потере реентерабельности рассматриваемой программы; однако в большинстве версий Кобола использование оператора ALTER совершенно определенно исключает это свойство программы. Что касается Фортрана и Кобола, то в действительности все это рассмотрение носит до некоторой степени академический характер, поскольку до настоящего времени мало кто из разработчиковЭВМ поставляет компиляторы с языков Фортран и Кобол, генерирующих реентерабельный *объектный код,*-— сами компиляторы и рабочие программы операционных систем, в которые они входят, могут быть реентерабельными, но Фортран- и Кобол-объектный код обычно этим свойством не обладают.

Другое нежелательное свойство модифицируемых программ состоит, в том, что обычно они не могут исполняться несколько раз подряд. Если мы хотим выполнить повторно модифицируемую во время работы программу, то нам придется повторить заново загрузку исходной программы. В этом, конечно, нет необходимости в том случае, если программист предусмотрел соответствующее автоинициирование своей программы. Заметим, что с той же трудностью можно столкнуться и при задании значений *переменных:* если программист пользуется оператором DATA в Фортране или предложением VALUE в Коболе, то всякое исполнение программы требует повторной загрузки исходной программы. (В ПЛ/1 это обычно не так: переменные, описанные с помощью атрибута INITIAL, получают свои начальные значения при входе в блок, содержащий их описание; это, однако, требует дополнительных ресурсов.)

Наше последнее критическое замечание по поводу модифицируемых программ имеет более практический характер и ближе к предмету этой главы: такая программа *значительно* труднее в понимании и отладке. Это в особенности верно в случае очень больших программ. Команда, модифицирующая код программы, может быть расположена в ее начале, а модифицируемые команды — совсем в другой части программы. Таким образом, в процессе отладки программист может локализовать ошибку, сужая анализируемый фрагмент программы до отдельной подпрограммы, однако если в такой подпрограмме есть команды, модифицированные операциями в каких-то других частях программы, и если листинг программы содержит несколько сот страниц, то может оказаться практически невозмож-

ным установить, *каким же образом* изменяется эта ошибочная подпрограмма.

Если плох оператор ALTER, то что можно сказать о других формах модификации программ? Соотнося возможные случаи (и имея в виду наше обсуждение структурного программирования в гл. 4), я бы сравнил использование программистами *вычисляемого* оператора GO TO (или GO TO DEPENDING ON в Коболе)с мелким правонарушением (если не имеется в виду сознательное его применение как средства моделирования Алгол-оператора CASE — одной из допустимых форм «черного ящика», рассмотренных в гл. 4); использование *назначенного* оператора GO TO — с уголовно наказуемым действием, применение оператора ALTER — с непредумышленным убийством, а написание модифицируемых программ на языке ассемблера — с действиями матерого убийцы. Хотя использование вычисляемого GO TO приводит к тому, что программа становится неструктурированной и несколько более трудной в отладке, мы по крайней мере можем перечислить все способы, которыми *могла бы исполняться* программа. В случае *назначенного* оператора GO TO (или использования в управляющих операторах в ПЛ/1 переменных типа метки) нам часто не удается опеределить по одной только записи оператора GO TO, в какие части программы может передаваться управление. То же самое в основном можно сказать и относительно оператора ALTER, причем здесь имеется тот дополнительный недостаток, что этот оператор вызывает нереентерабельность программы (хотя, справедливости ради, стоит заметить, что эту трудность можно было бы разрешить средствами компилятора Кобол-программ). Модифицированные команды в программах на языке ассемблера следует считать наибольшим злом, поскольку в этом языке программисту предоставляются практически неограниченные возможности реализации модифицируемых программ.

Здесь логично задать следующий вопрос: если самомодифицирующиеся программы обладают столь многими недостатками, то почему программистам так нравится этот подход? Обычно в ответ можно услышать о соображениях удобства и эффективности. Программисту часто кажется проще и легче реализовать определенные программные ситуации с помощью операторов ALTER или назначенного GO TO; кроме того, программисту часто кажется, что в такой реализации программа будет исполняться *быстрее,* чем в любой другой. Однако обычно использование модифицируемых программ кажется более удобным лишь потому, что программист недостаточно думал о поисках другого способа решения задачи. Если программисту представляется, что программирование с операторами ALTER и назначенным GO TO более удобно, то, как правило, это объясняется тем, что ему не удалось учесть те дополнительные потери машинного времени, возникающие при отладке такой его программы. Существуют лучшие способы написания программ, и часто они ока-

```
* ЭТА ПОДПРОГРАММА ОТПЕЧАТАЕТ ВСЕ СЛУЧАИ ВХОЖДЕНИЯ ИЛИ
  НЕВХОЖДЕНИЯ ДАННОГОЭЛЕМЕНТАВДАННУЮОБЛАСТЬПАМЯТИ.
* ПОРЯДОК ОБРАЩЕНИЯ ПРИВОДИТСЯ НИЖЕ:
*          BAL   R14, WSEARCH   ПОИСК ВСЕХ ВХОЖДЕНИЙ ЭЛЕ-
                                  МЕНТА
*          BAL   R14, NSEARCH   ПОИСК ВСЕХ НЕВХОЖДЕНИЙ ЭЛЕ-
                                  МЕНТА
* R1 СОДЕРЖИТ АДРЕС ПЕРВОГО СЛОВА ПРОСМАТРИВАЕМОЙ ОБ-
  ЛАСТИ ПАМЯТИ
* R2 СОДЕРЖИТ ПОЛОЖИТЕЛЬНОЕ ЦЕЛОЕ — ЧИСЛО СЛОВ, ПОДЛЕ-
  ЖАЩИХ ПРОВЕРКЕ
* R3 СОДЕРЖИТ ОТЫСКИВАЕМЫЙ ЭЛЕМЕНТ
* „МАСКА" ОПРЕДЕЛЯЕТ ТЕ РАЗРЯДЫ, КОТОРЫЕ ИСПОЛЬЗУЮТСЯ
  ПРИ СРАВНЕНИИ:
*ЕСЛИ N-Й РАЗРЯД МАСКИ РАВЕН 1, TON-ЙРАЗРЯДСОДЕРЖИ-
  МОГО R3
* И СЛОВА ИЗ ОБЛАСТИ ПАМЯТИ БУДУТ СРАВНИВАТЬСЯ НА РАВЕН-
  СТВО (ИЛИ НЕРАВЕНСТВО)
WSEARCH  L     R4,SWITCH1    ВЫБРАТЬ КОМАНДУ СРАВНЕНИЯ
         B     SEARCH        ВОЙТИ В РАЗДЕЛ ОБЩИХ КОДОВ
NSEARCH  L     R4,SWITCH2    ВЫБРАТЬДРУГУЮКОМАНДУСРАВ-
                             НЕНИЯ
SEARCH   ST    R4,COMPARE    ФИКСИРОВАТЬ КОМАНДУ СРАВНЕ-
                             НИЯ
         N     R3,MASK       СМОТРЕТЬ ТОЛЬКО РАЗРЯДЫ, ОП-
                             РЕДЕЛЯЕМЫЕ МАСКОЙ
         L     R4,ZERO       НАЧАЛЬНОЕ ЗНАЧЕНИЕ СЧЕТЧИ-
                             КА СЛОВ ПАМЯТИ
LOOP     L     R5,(R4,R1)    ВЫБРАТЬ СЛОВО ИЗ ПАМЯТИ,
                             ИСПОЛЬЗУЯ R1 КАК БАЗУ
         N     R5,MASK       СМОТРЕТЬ ТОЛЬКО РАЗРЯДЫ, ОП-
                             РЕДЕЛЯЕМЫЕ МАСКОЙ
         CR    R5,R3         СРАВНЕНИЕДВУХЭТИХЗНАЧЕНИЙ
COMPARE  BE    FOUND         ***ЭТАКОМАНДА—ИЗМЕНЯЕМАЯ***
NEXT     LA    R4,4(R4)      ПРИРАЩЕНИЕ ЗНАЧЕНИЯ СЧЕТ-
                             ЧИКА ПАМЯТИ
         BCT   R2,LOOP       ПОВТОРИТЬ ЦИКЛ НА СЛЕДУЮЩЕМ
                             СЛОВЕ
         BR    R14           ВЫХОД
* ВОЙТИ ЗДЕСЬ В СЛУЧАЕ УСПЕШНОГО СРАВНЕНИЯ
FOUND  * L     R5,(R4,R1)    СНОВА ВЫБРАТЬ СЛОВО ИЗ ПА-
                             МЯТИ
         STM   R1,R5,TABLE   УПРЯТАТЬ ЗНАЧЕНИЯ РЕГИСТРОВ
         LR    R1,R4         ПОСЛАТЬ АДРЕС СЛОВА ПАМЯТИ
                             В R1
         BAL   R13,PRINT     ОБРАЩЕНИЕКПОДПРОГРАММЕПЕ-
                             ЧАТИ АДРЕСА
       . L     R1,TABLE+4    ВЫБРАТЬ СОДЕРЖИМОЕ СЛОВА
                             ПАМЯТИ
         BAL   R13,PRINT     НАПЕЧАТАТЬ СОДЕРЖИМОЕ СЛОВА
         LM    R1,R5,TABLE   ВОССТАНОВИТЬЗНАЧЕНИЯРЕГИСТ-
                             РОВ
         B     NEXT          ВЕРНУТЬСЯ К ПРОВЕРКЕ СЛЕДУЮ-
                             ЩЕГО СЛОВА
SWITCH1  BE    FOUND         ***ЭТО ИСПОЛЬЗУЕТСЯПРОГРАМ-
                             МОЙ WSEARCH ***
```

```
SWITCH2   BNE   FOUND      *** ЭТО ИСПОЛЬЗУЕТСЯ ПРОГРАМ-
                           МОЙ NSEARCH ***
MASK      DS    F          СЛОВО МАСКИ
ZERO      DC    F'0'
TABLE     DS    5F         ПАМЯТЬ ДЛЯ СОХРАНЕНИЯ РЕ-
                           ГИСТРОВ
```

Рис. 5.1а. Программа поиска в области памяти, написанная на языке ассемблера Системы IBM/360.

зываются столь же удобными; в большинстве случаев они столь же и эффективны.

Проиллюстрируем эту точку зрения некоторыми примерами. Наиболее широко метод модифицируемых программ используется для придания некоторому модулю функций по решению нескольких похожих задач. Так, на рис. 5.1а показана подпрограмма на языке ассемблера Системы IBM/360, которая может использоваться для поиска в заданной области памяти всех случаев *вхождения* или всех случаев *невхождения* определенного элемента. Простым изменением одной команды условного перехода программист обеспечивает использование этой подпрограммы для решения двух различных, хотя и похожих, задач. С точки зрения минимизации машинного времени с большой уверенностью можно сказать, что решение, показанное на рис. 5.1а, достаточно эффективно. С другой стороны, подпрограмма, показанная на рис. 5.1б, чуть менее эффективна, но обладает всеми описанными выше достоинствами: она реентерабельна допускает повторное использование, проще в отладке и для понимания.

На рис. 5.2а показана аналогичная ситуация в программировании на ПЛ/1. В зависимости от результатов некоторых сравнений программист желает выполнить слегка отличающиеся последовательности действий. Хотя такой метод кодирования, может быть, проще, подход с использованием операторов IF-THEN, показанный на рис. 5.2б, представляет собой значительно лучший способ действий (хотя он и может привести к несколько менее эффективному программному коду в зависимости от особенностей используемого компилятора). Рис. 5.3а и рис. 5.3б иллюстрируют аналогичную ситуацию в Фортране; рис. 5.4а и рис. 5.4б показывают ту же задачу на Коболе. Во всех приведенных примерах предпочтительная реализация решения может потребовать несколько большего времени ЦП и несколько большего внимания программиста к предварительному планированию. Должно быть очевидным, однако, что результирующая программа оказывается *значительно* лучше.

Короче говоря, мы видим, что различные формы модифицируемых программ обычно используются для обработки *специальных*

```
* ЭТА ПОДПРОГРАММА ВЫПОЛНЯЕТ ТУ ЖЕ ОБРАБОТКУ, ЧТО И
* ПОДПРОГРАММА РИС. 5.1А
* ОНА ВЫЗЫВАЕТСЯ ТОЧНО ТАК ЖЕ И ИСПОЛЬЗУЕТ ТЕ ЖЕ СО-
* ГЛАШЕНИЯ ОТНОСИТЕЛЬНО АРГУМЕНТОВ. ОДНАКО ОНА НЕ МО-
* ДИФИЦИРУЕТСЯ ВО ВРЕМЯ ИСПОЛНЕНИЯ

WSEARCH  LA    R6,0          ИСПОЛЬЗОВАТЬИ6ДЛЯСЧЕТЧИКА
                              ТАБЛИЦЫ
         B     SEARCH        ВОИТИ В РАЗДЕЛ ОБЩИХ КОДОВ
NSEARCH  LA    R6,4          УСТАНОВИТЬ R6 ДЛЯ ВЫБОРА ДРУ-
                              ГОЙ КОМАНДЫ
SEARCH   N     R3,MASK       СМОТРЕТЬ ТОЛЬКО РАЗРЯДЫ, ОП-
                              РЕДЕЛЯЕМЫЕ МАСКОЙ
         L     R4,ZERO       НАЧАЛЬНОЕЗНАЧЕНИЕСЧЕТЧИКА
                              СЛОВ ПАМЯТИ
LOOP     L     R5,(R4,R1)    ВЫБРАТЬ СЛОВО ИЗ ПАМЯТИ, ИС-
                              ПОЛЬЗУЯ R1 КАК БАЗУ
         N     R5,MASK       СМОТРЕТЬ ТОЛЬКО РАЗРЯДЫ, ОП-
                              РЕДЕЛЯЕМЫЕ МАСКОЙ
         CR    R5,R3         СРАВНЕНИЕ ЭТИХ ДВУХ ЗНАЧЕ-
                              НИЙ
         EX    O,SWITCH(R6)  ИСПОЛНИТЬ СООТВЕТСТВУЮЩУЮ
                              КОМАНДУ ПЕРЕХОДА
NEXT     LA    R4,4(R4)      ПРИРАЩЕНИЕ ЗНАЧЕНИЯ СЧЕТ-
                              ЧИКА ПАМЯТИ
         BCT   R2,LOOP       ПОВТОРИТЬ ЦИКЛ НА СЛЕДУЮ-
                              ЩЕМ СЛОВЕ
         BR    R14           ВЫХОД
* ВОЙТИ ЗДЕСЬ В СЛУЧАЕ УСПЕШНОГО СРАВНЕНИЯ
FOUND    L     R5,(R4,R1)    СНОВА ВЫБРАТЬ СЛОВО ИЗ ПА-
                              МЯТИ
         STM   Rl,R6,TABLE   УПРЯТАТЬ ЗНАЧЕНИЯ РЕГИСТРОВ
         LR    R1,R4         ПОСЛАТЬ АДРЕС СЛОВА ПАМЯТИ
                              В R1
         BAL   R13,PRINT     НАПЕЧАТАТЬ АДРЕС СЛОВА
         L     Rl,TABLE + 4  СНОВА ВЫБРАТЬ СОДЕРЖИМОЕ
                              СЛОВА ПАМЯТИ
         BAL   R13,PRINT     НАПЕЧАТАТЬЕГО
         LM    Rl,R6,TABLE   ВОССТАНОВИТЬ ЗНАЧЕНИЯ РЕ-
                              ГИСТРОВ
         B     NEXT          ВЕРНУТЬСЯКПРОВЕРКЕСЛЕДУЮ-
                              ЩЕГО СЛОВА
*
* КОНСТАНТЫ И ДАННЫЕ-ДЛЯ РЕЕНТЕРАБЕЛЬНОСТИ ИХ МОЖНО
* БЫЛО БЫ ПОМЕСТИТЬ В РАЗДЕЛЕ ДАННЫХ

ZERO     DC    F'0"
MASK     DS    F             СЛОВО МАСКИ
TABLE    DS    BF            ПАМЯТЬ ДЛЯ СОХРАНЕНИЯ РЕ-
                              ГИСТРОВ
SWITCH   BE    FOUND         ЭТО БУДЕТ ИСПОЛНЕНО ЕСЛИ
                              R6 = 0
         BNE   FOUND         ЭТО БУДЕТ ИСПОЛНЕНО ЕСЛИ
                              R=1
```

Рис. 5.1б. Слегка измененная версия той же программы для Системы IBM/360.

```
          DECLARE ORDERFILE OUTPUT:
          DECLARE SHIPMENTFILE OUTPUT;
          DECLARE CANCELFILE OUTPUT;
          DECLARE ERRORFILE OUTPUT;
          DECLARE (TRANSCODE,DOLLAR.TOTALQ,T0TALD,QUANTITY) FIXED;
          DECLARE SWITCH LABEL;
          TOTALQ = 0;
          TOTALD = 0;
LOOP:     . SWITCH = ERROR;
          GET LIST (TRANSCODE, QUANTITY, DOLLAR);
          IF TRANSCODE = 1 THEN SWITCH = ORDER;
          IF TRANSCODE = 3 THEN SWITCH = SHIPMENT;
          IF TRANSCODE »= Б THEN SWITCH » CANCEL:
          GO TO SWITCH;
ERROR:    PUT FILE (ERRORFILE) LIST (TRANSCODE,QUANTITY,DOLLAR):
          GO TO LOOP;
ORDER:    TOTALQ = TOTALQ + QUANTITY;
          TOTALD = TOTALD + DOLLAR;
          PUT FILE (ORDERFILE) LIST (TRANSCODE.QUANTITY.DOLLAR);
          GO TO LOOP;
CANCEL:   TOTALQ « TOTALQ - QUANTITY;
          TOTALD = TOTALD - DOLLAR;
          PUT FILE (CANCELFILE) UST (TRANSCODE,QUANTITY,DOLLAR):
          GO TO LOOP;
SHIPMENT: PUT FILE (SHIPMENTFILE) LIST (TRANSC0DE,QUANTITY,DOLLAR):
          GO TO LOOP;
```

Примечание: Эта программа предназначена для чтения карт сообщений и обработки их в соответствии с *кодом сообщения.* В этом крайне простом примере довольно легко уяснить смысл применения переменных типа метки операторов; в более реальном варианте этой программы действие этих переменных было бы менее ясным и легко скрывалось бы среди тысяч других операторов.

Рис. 5.2а. Программа на ПЛ/1, использующая переменные типа метка оператора.

случаев в *условиях ветвления логики,* а также в ситуациях, требующих выполнения «сходных» процессов. Из приведенного выше обсуждения вам должно быть ясным, что многие недостатки модифицируемых программ обычно не компенсируются небольшим выигрышем в машинном времени и сэкономленной памяти; вам должно быть также ясно, что для таких средств программирования, как оператор ALTER в Коболе и назначенный оператор GO TO в Фортране, существуют хорошие альтернативы. Наилучший с точки зрения модульности способ состоит, по-видимому, в оформлении каждого специального случая в виде замкнутой подпрограммы; именно такой подход отражен на рис. 5.46. Во многих случаях не хуже оказывается и решение, основанное на применении IF-THEN, что

```
          DECLARE ORDERFILE OUTPUT;
          DECLARE SHIPMENTFILE OUTPUT;
          DECLARE CANCELFILE OUTPUT;
          DECLARE ERRORFILE OUTPUT;
          DECLARE (TRANSCODE,DOLLAR,TOTALQ,TOTALD,QUANTITY) FIXED;
          TOTALQ = 0;
          TOTALD = 0;
LOOP:     GET LIST (TRANSCODE.QUANTITY.DOLLAR);
          IF TRANSCODE = 1 THEN
               DO;
               TOTALQ = TOTALQ + QUANTITY;
               TOTALD = TOTALD + DOLLAR;
               PUT FILE (ORDERFILE) LIST (TRANSCODE,QUANTITY,DOLLAR):
               END;
          ELSE
               IF TRANSCODE = 3 THEN
                    PUT FILE (SHIPMENTFILE) LIST (TRANSCODE.QUANTITY.DOLLAR);
               ELSE
                    IF TRANSCODE = 5 THEN
                         DO;
                         TOTALQ = TOTALQ - QUANTITY;
                         TOTALD = TOTALD - DOLLAR;
                         PUT FILE (CANCELFILE) LIST (TRANSCODE.QUANTITY.DOLLAR);
                         END;
                    ELSE PUT FILE (ERRORFILE) LIST (TRANSCODE.QUANTITY.DOLLAR);
          GO TO LOOP;
```

Рис. 5.26. Слегка измененный вариант той же программы на ПЛ/1.

иллюстрируется рис. 5.2б. К сожалению, как мы видели в гл. 4, кажется лишь ПЛ/1 и Алгол располагают блочной структурой BEGIN-END, необходимой для реализации хоть отчасти изящного подхода, отличного от тривиального IF-THEN-построения, хотя язык КОБОЛ в этом отношении обычно достаточен. Распознав специальный случай в некоторой точке нашей программы, мы можем использовать команду безусловной передачи управления в некоторый специальный фрагмент программного кода, как это было показано на рис. 5.3б в примере в Фортране; однако такая структура программы намного менее желательна, так как при этом значительно снижается ясность программы. В подобной ситуации использование вычисляемого оператора GO TO следует рассматривать как меньшее зло. Наконец, мы можем использовать в программе логические признаки и многие другие приемы на языке ассемблера, как это показано на рис. 5.16.

```
C
C   THIS PROGRAM ACCOMPLISHES THE SAME PURPOSE AS THE PL/I PROGRAM
C     IN FIGURE 5.2(A)
C
      INTEGER TCODE
      TOTALQ=O
      TOTALD=0
10    ASSIGN 20 TO SWITCH
      READ 100,TCODE,QUANT,DOLLAR
      IF (TCODE .EQ. 1) ASSIGN 30 TO SWITCH
      IF (TCODE .EQ. 3) ASSIGN 40 TO SWITCH
      IF (TCODE .EQ. 5) ASSIGN 50 TO SWITCH
      GO TO SWITCH
C
C     ERROR ROUTINE
C
20    WRITE (3,200) TCODE,QUANT,DOLLAR
      GO TO 10
C
C     ORDER-HANDLING ROUTINE
C
30    TOTALQ=TOTALQ+QUANT
      TOTALD=TOTALD + DOLLAR
      WRITE (4,300) TCODE,QUANT,DOLLAR
      GO TO 10
C
C     CANCELLATION ROUTINE
C
40    TOTALQ=TOTALQ-QUANT
      TOTALD=TOTALD-DOLLAR
      WRITE (5,400) TCODE,QUANT,DOLLAR
      GO TO 10
C
C     SHIPMENT ROUTINE
C
50    WRITE (6,500) TCODE,QUANT,DOLLAR
      GO TO 10
```

Рис. 5.3а. Фортран-программа, использующая назначенные операторы GO TO.

```
      INTEGER TCODE
      TOTALQ=0
      TOTALD=O
10    READ 100, TCODE,QUANT,DOLLAR
      IF (TCODE .EQ. 1) GO TO 30
      lF (TCODE .EQ. 3) GO TO 40
      IF (TCODE .EQ. Б) GO TO 5»
C
C     FALL THROUGH TO THE ERROR ROUTINE
C
20    WRITE (3,200) TCQDE,QUANT,OOLLAR
      GO TO 10
C
C     ORDER-HANDLING ROUTINE
C
30    TOTALQ=TOTALQ+QUANT
      TOTALD =TOTALD + DOLLAR
      WRITE (4,300) TCODE,QUANT,OOLUn
      GO TO 10
C
C     CANCELLATION ROUTINE
C
40    TOTALQ=TOTALQ-QUANT
      TOTALD =TOTALD - DOLLAR
      WRITE (5,400) TCODE,aUANT,09LWR
      GO TO 10
C
C     SHIPMENT ROUTINE
C
50    WRITE (6,500) TCODE,QUANT.DOLLAR
  .,, GO TO 10
```

Рис. 5.3б. Слегка измененный вариант той же Фортран-программы.

```
      MOVE 0 TO TOTALQ.
      MOVE 0 TO TOTALD.
GET-INPUT. READ TRANSACTION-CARDS.
      IF TRANSACTION-CODE=1 THEN ALTER SWITCH-PARAGRAPH TO
         PROCEED TO ORDER-ROUTINE
      ELSE IF TRANSACTI0N-C0DE=3 THEN ALTER SWITCH-PARAGRAPH TO
         PROCEED TO CANCEL-ROUTIN6
      ELSE IF TRANSACTION-CODE=5 THEN ALTER SWITCH-PARAGRAPH TO
         PROCEED TO SHIPMENT-ROUTINE
      ELSE ALTER SWITCH-PARAGRAPH TO PROCEED TO ERROR-ROUTINE.
SWITCH-PARAGRAPH. GO TO ERROR-ROUTINE.
ERROR-ROUTINE.
      MOVE TRANSACTION-CODE TO ERROR-CODE.
      MOVE QUANTITY TO ERROR-QUANTITY.
      MOVE DOLLAR TO ERROR-DOLLAR.
      WRITE ERROR-FILE.
      GO TO GET-INPUT.
ORDER-ROUTINE.
      COMPUTE TOTALQ=TOTALQ+QUANTITY.
      COMPUTE TOTALD=TOTALD + DOLLAR.
      MOVE TRANSACTION-CODE TO ORDER-CODE.
      MOVE DOLLAR TO ORDER-DOLLAR.
      MOVE QUANTITY TO ORDER-QUANTITY.
      WRITE ORDER-FILE.
      GO TO GET-INPUT.
CANCEL-ROUTINE.
      COMPUTE TOTALQ=TOTALQ-QUANTITY.
      COMPUTE TOTALD=TOTALD-DOLLAR.
      MOVE TRANSACTION-CODE TO CANCEL-CODt.
      MOVE DOLLAR TO CANCEL-DOLLAR.
      MOVE QUANTITY TO CANCEi-QUANTITY.
      WRITE CANCEL-FILE.
      GO TO GET-INPUT.
SHIPMENT-ROUTINE.
      MOVE TRANSACTION-CODE TO SHIPMENT-COD£.
      MOVE DOLLAR TO SHIPMENT-DOLLAR.
      MOVE QUANTITY TO SHIPMENT-QUANTITY.
      WRITE SHIPMENT-FILE.
      GO TO GET-INPUT.
```

Рис. 5.4а, Кобол-программа с операторами ALTER,

```
      MOVE 0 TO TOTALQ.
      MOVE 0 TO TOTALD.
GET-INPUT. READ TRANSACTION-CARDS.
      IF TRANSACTION-CODE=1 THEN PERFORM ORDER-ROUTINE
      ELSE IF TRANSACTION-CODE=3 THEN PERFORM CANCEL-ROUTINE
      ELSE IF TRANSACTION-CODE=5 THEN PERFORM SHIPMENT-ROUTINE
      ELSE PERFORM ERROR-ROUTINE.
      GO TO GET-INPUT.
ERROR-ROUTINE.
      MOVE TRANSACTION-CODE TO ERROR-CODE.
      MOVE QUANTITY TO ERROR-QUANTITY.
      MOVE DOLLAR TO ERRQR-DOLLAR.
      WRITE ERROR-FILE.
      EXIT.
ORDER-ROUTINE.
      COMPUTE TOTALQ=TOTALQ+QUANTITY.
      COMPUTE TOTALD=TOTALD + DOLLAR.
      MOVE TRANSACTION-CODE TO ORDER-CODE.
      MOVE DOLLAR TO ORDER-DOLLAR.
      MOVE QUANTITY TO ORDER-QUANTITY.
      WRITE ORDER-FILE.
      EXIT.
CANCEL-ROUTINE.
      COMPUTE TOTALQ=TOTALQ-QUANTITY.
      COMPUTE TOTALD=TOTALD-DOLLAR.
      MOVE TRANSACTION-CODE TO CANCEL-CODE.
      MOVE QUANTITY TO CANCEL-QUANTITY.
      MOVE DOLLAR TO CANCEL-DOLLAR.
      WRITE CANCEL-FILE.
      EXIT.
SHIPMENT-ROUTINE.
      MOVE TRANSACTION-CODE TO SHIPMENT-CODE.
      MOVE DOLLAR TO SHIPMENT-DOLLAR.
      MOVEL QUANTITY TO SHIPMENT-QUANTITY.
      WRITE SHIPMENT-FILE.
      EXIT.
```

Рис. 5.46. Слегка измененный вариант той же Кобол-программы.

5.2. Дополнительные методы повышения читабельности программ

Одним из качеств хорошего программиста можно считать его способность удерживаться от соблазна использовать ухищренные приемы программирования. Этот принцип не очень важен в программировании на Фортране и Коболе ввиду относительной ограниченности этих языков; однако в гибких языках высокого уровня типа ПЛ/1 или Алгола программист часто имеет возможность писать чрезвычайно неясные, нестандартные и усложненные программные коды. В языке ассемблера этот вопрос стоит еще острее: в машинном коде имеется практически бесконечное число возможных способов программирования одной и той же задачи. По-видимому, было бы несправедливым прибегать к широким обобщениям подобных ситуаций, поскольку в определенных условиях могут быть оправданы даже самые специфические особенности данного языка программирования. Мы пытаемся лишь выразить наше *отношение* к этому, которое можно сформулировать так: «Вам всегда следует использовать самые простые, наименее запутанные, наименее специфические средства языка, позволяющие должным образом решить рассматриваемую задачу. Не прибегайте к усложненному программированию только ради того, чтобы похвастаться перед своими коллегами, что вам все-таки удалось употребить команду XYZ!»

Мы можем несколько развить высказанный принцип. Не тратьте свое время на изучение программы ваших коллег, пытаясь найти в них изолированные фрагменты программного кода. Не ищите неприятностей, пытаясь переписать чью-то сравнительно хорошо составленную 100-командную программу, чтобы получить чуть более умную 99-командную программу. Быть может, вам удастся сэкономить немного времени ЦП и немного памяти; но, с другой стороны, дополнительное время на программирование, компиляцию и тестирование часто может нивелировать эти выигрыши. Эту мысль часто бывает очень трудно довести до понимания некоторых программистов, особенно из числа молодых, блестящих и наиболее агрессивных. Часто кажется, что высшее наслаждение в жизни они находят в кропотливом изучении математического обеспечения разработчика ЭВМ и программ своих коллег (которых они обычно считают слишком старыми, неповоротливыми, слишком невежественными и слишком небрежными, чтобы создавать по-настоящему *хорошие* программы) с целью отыскания различных неэффективностей незначительного характера. Они могут высказаться критически о том, что коллега использовал три команды для вычисления функций XYZ, а он мог бы это сделать с помощью одной команды, или что в программе переменной ABC присваивается нулевое значение, в то время как она *уже* имеет это значение. Затем эти «бездельники от программирования», как их выразительно называют в некоторых

организациях, добьются сокращения на десять микросекунд времени исполнения программы, потратят десять минут на компиляцию программы с их улучшениями, не удосужатся внести каких-либо объяснений по поводу внесенных изменений и без всякой нужды вызовут к себе враждебное отношение со стороны коллег на несколько следующих недель.

Как было отмечено в первых четырех главах этой книги, программисты часто не склонны использовать ряд здравых правил программирования. Так, они редко применяют нисходящую схему проектирования, предложенную в гл. 2, хотя с интуитивной точки зрения это наиболее организованный подход к проектированию, кодированиюитестированию программ. Программисты редко *на практике* реализуют принцип модульности, хотя многие из них его проповедуют. Мало кто их программистов придерживается принципа использования стандартных подпрограмм, хотя они и соглашаются, что обычно это хорошая практика. Аналогично очень немногие программисты предпринимают сознательные усилия для обеспечения простоты их программ, даже если они и *говорят,* что сделали бы все возможное, чтобы минимизировать время, необходимое для их отладки.

В чем же дело? Кажется, что причина в основном относится к области психологии: даже если программист осознает, что он делает программу более трудной в отладке и менее понятной, он не в состоянии превозмочь стремление к усложнению и применению специфических решений. Разновидностью этого может служить такое рассуждение программиста: «Если бы не предполагалось использовать эти (специфические) команды, то разработчик машины не включал бы их в систему команд,— если вы не применяете их, то вы не пользуетесь всеми возможностями, которые заложены в машине!» Иногда можно столкнуться и с диаметрально противоположной точкой зрения: программист упорно отказывается верить, что сложные многоцелевые программы, насыщенные ухищренными приемами программирования и модифицируемые в процессе исполнения, могут быть трудными в отладке; иногда можно встретить программиста, заявляющего: «Всякий хоть чего-нибудь стоящий программист должен понять такую программу!» Иногда программист прибегает к подобным сомнительным приемам, чтобы поднять свою репутацию в глазах коллег: «Ну да,— говоритон своим ошарашенным друзьям,— я записал решение этой задачи о начислении зарплаты в виде рекурсивной процедуры; это делает подпрограмму вычисления налогов значительно более изящной!» Эта поза обычно исчезает, когда программист становится более опытным и зрелым; к сожалению, зрелость не является универсальной чертой характера, и она не всегда растет пропорционально опыту. Поэтому руководитель программных разработок должен проявлять бдительность в отношении подобной позиции своих программистов.

Иногда находятся *обоснованные* доводы против простейших решений; их важнейшим потенциальным недостатком является неэффективность. «Незамысловатая» программа может не использовать всех возможности развитого языка программирования или вычислительной машины; в результате программа может либо медленнее исполняться, либо занимать больше памяти. Это было очевидным в примерах, показанных на рис. 5.1б, 5.2б, 5.3б и 5.4б. Здесь, конечно, необходимо установить некоторые эквиваленты ценностей, однако большинство программистов не проводят соответствующего тщательного анализа; они не отдают себе отчет в том, что в большинстве программных разработок узким местом является *недостаток времени на программирование* и *отладку,* а не потребности программы в машинном времени и памяти. Если простейшее решение действительно требует чрезмерных количеств времени ЦП и (или) памяти, то у нас может появиться желание прибегнуть к некоторым ухищрениям; однако, как мы уже установили в этой главе, требование простоты обычно не приводит к существенному снижению эффективности.

Обратите внимание на тонкое различие между принципом простоты в *проектировании* и принципом простоты в *программировании.* Вам следует добиваться того, чтобы ваш проект был как можно более совершенным; он должен быть модульным.обладатьбольшей общностью и с наибольшей точностью отвечать вашей задаче. Однако, когда вы приступаете к написанию программ, ваши решения должны быть максимально простыми. Существует ряд специальных приемов программирования, перечисляемых ниже, которые должны служить средством упрощения результирующей программы и повышения ее читабельности; возможно, вы добавите к ним ваши собственные рекомендации.

5.2.1. Избегайте неоправданно сложных арифметических выражений

Фактически все языки программирования высокого уровня, а также ряд языков ассемблера позволяют программисту формировать *арифметические выражения;* в каждом таком языке определено некоторое правило *старшинства операций,* позволяющее исключить неоднозначность выражений в бесскобочной записи. Так, большинство программистов при чтении операторной фразы

$$A = B * C + 3$$

согласилось бы с тем, что сначала выполняется, умножение B на C, затем к результату прибавляется целое 3.

К сожалению, некоторые программисты — особенно из среды научных работников — позволяют себе довольно большую свободу

в этом отношении. Читая фразу

```
A = B*C + 3**D/-3*X+Y*4
```

вы даже не уверены в том, что она верна синтаксически (хотя можно надеяться, что компилятор смог бы это установить). Во всяком случае, совсем не ясно, какие здесь выполняются операции. Весьма вероятно, что программист может ошибочно представить себе порядок исполнения операций в данном конкретном языке (в таком случае он не получит ожидаемого результата); тем более вероятно, что при толковании смысла этого выражения другие программисты могут просто запутаться.

Очевидным выходом из этого затруднения является внесение некоторых осмысленных *скобок,* разделяющих соответствующие подвыражения и разъясняющих последовательность и природу выполняемых арифметических операций. Такие выражения на самом деле избыточны в том смысле, что арифметические выражения можно записывать и без скобок, и это, кажется, беспокоит некоторых программистов, которые чувствуют, что мировой запас скобок ограничен и их следует беречь на те редкие случаи, когда они абсолютно необходимы! Прагматически настроенный программист, конечно, понимает, что лишние скобки ничтожно замедляют процесс компиляции и не влияют на время исполнения скомпилированной программы (хотя, быть может, это зависит от компиляторов). Кто-то может настаивать даже на том, что скобки полезно употребить и в *первом* из приведенных выше примеров, т. е. что было бы неплохо записать

```
A=(B*C)+3
```

В случае выражения столь сложного, как в приведенном выше втором примере, вообще неясно, сможет ли внесение скобок сделать его более «читабельным». Проще ли действительно следующая форма записи нашего примера?

```
A=(B*C)+(((3**D)/(-3))*X+(Y*4)
```

В таких случаях может быть намного разумнееразбить исходное выражение на несколько более простых выражений:

```
PART1=B*C
PART2 = X* ((3 ** D)/(-3))
PART3 = Y*4
A = PART1 +PART2 + PART3
```

Приведенный выше пример записи может быть и не самой «читабельной» формой представления исходного сложного выражения. Вполне вероятно, что такая запись приведет к менее эффективному объектному коду, если компилятор не оснащен аппаратом глобальной оптимизации. Она отражает, однако, попытку программиста представить читателю его вычисления более четко.

5.2.2. Компонуйте вложенные условные операторные фразы

Интересно отметить, что стандартными руководствами по программированию на Коболе во многих организациях специально оговаривается запрет на использование вложенных операторов IF-THEN-ELSE типа

```
IF A1 THEN B1 ELSE IF A2 THEN IF A3 THEN B2 ELSE B3 ELSEB4
```

Не следует удивляться тому, что в подобной записи такая операторная фраза трудна для понимания; в большинстве случаев вложенные условные операторные фразы оказываются довольно «читабельными», если их записывать в следующей форме:

```
IF A1 THEN B1
ELSE IF A2
     THEN IF A3 THEN B2
               ELSE B3
     ELSE B4
```

Как и в случае использования скобок в сложных выражениях, эта идея ступенчатой (или хорошо оформленной) организации составных условных операторных выражений может увести слишком далеко. Однако, если имеется всего лишь три или четыре уровня условных операторов (как в приведенном выше примере), аккуратное оформление и ступенчатая организация вашей программы были бы достаточными средствами разъяснения ее смысла другим читателям. Те же замечания справедливы, конечно, и в отношении таких языков, как ПЛ/1 и Алгол, но только не Фортран!

5.2.3. Организуя листинги ваших программ, старайтесь сделать их более «читабельными»

В предыдущем разделе мы отметили, что соответствующей организацией вложенных условных операторных фраз можно сделать их болеепонятными. Вдействительности это лишь один пример некоторой более общей рекомендации: *листинг вашей программы должен быть привлекательным на вид, чтобы читателю было легко — даже интересно — его изучать.* Некоторые программисты полагают, что это скорее дело операторов по перфорации программ. Другие, работающие на таких языках, как Лисп и ПЛ/1 (и в меньшей степени Фортран, Кобол и Алгол), с успехом пользуются вспомогательными автоматизированными средствами оформления исходной программы, переупорядочения последовательности операторов и т. д. При использовании любых средств стоит иметь в виду следующие пожелания:

1. *В одной строке листинга программы не следует располагать больше одной операторной фразы исходной программы.* Хотя и верно,

что бумага представляет ценность и что деревья надо беречь (как мне однажды заметила одна группа серьезных программистов), я полагаю, что наши экологи от программирования могли бы приложить свои усилия в каком-нибудь более обещающем направлении (быть может, в отношении слишком многословных изданий по вычислительным наукам?). Однажды я имел сомнительную честь увидеть полный листинг довольно сложной операционной системы с разделением времени, занимающий около 15 листов; все 128 позиций листинга были заняты записями многочисленных операторов, разделенными точкой с запятой. В некоторых случаях оказываются более читабельными два или три оператора, расположенных в одной строке; в мини-ЭВМ типа PDP-8, например, применением оператора RTR можно за один раз сдвинуть вправо содержимое сумматора только на два разряда, таким образом, при необходимости сдвинуть содержимое сумматора вправо на шесть разрядов было бы разумным записать

```
RTR; RTR; RTR     /СДВИГ СУММАТОРА НА ШЕСТЬ БИТОВ ВПРАВО/
```

Аналогичные ситуации иногда встречаются в языках высокого уровня, и поэтому записи нескольких операторов могут быть расположены в одной строке, однако обычно программисты слишком неумеренно пользуются этой возможностью.

2. Полъзуйтесь «полем идентификации», которое обычно занимает колонки 73—80 образа перфокарты исходной программы. Первые четыре или иять символов могут использоваться для идентификации программы, а остальные могут служить порядковым номером; так, мы могли бы видеть такое содержание поля идентификации

```
GLOP0010
```

Между порядковыми номерами следует ввести интервал по крайней мерев 10, если не в 100, единиц, с тем чтобы в последующем можно было вставить новые операторы; такие вставки должны быть помечены идентификаторами ЗАПЛАТА или НОВЫЙ и т. д. Как мы отметили выше,обычно имеются различные пакеты программ по обеспечению сопровождения программ, позволяющие программисту изменять последовательность операторов в листинге программы. Крометого, многие компиляторы и ассемблеры проверкой поля идентификации устанавливают правильность расположения операторов в исходной программе. Эти методы, конечно, не представляют собой какого-то новшества или чуда; однако они работают и кажутся более разумными, чем подход, выбранный программистом на Фортране, который, нарушая и эту рекомендацию, и приведенную выше, записывает следующую последовательность Фортран-операторов:

```
10    A = B = C;  GO  TO  20;
20    X = Y * Z;  GO  TO  30;
30    Q/R;  GO  TO  40;
```

На вопрос, почему он включил так много явно избыточных операторов GO TO, программист ответил: «У меня было слишком много неприятностей из-за перемешивания карт в колоде. При таком подходе нужно лишь, чтобы правильно лежали только первая и последняя карты — все остальные можно располагать как попало!»

3. Если запись оператора занимает больше одной строки листинга программы, пользуйтесь переносом, сохраняющим читабельность программы. Не разбивайте, например, имя переменной или константы таким бессмысленным образом:

```
НОЙ = КОШКА + СОБАКА + АЛЛИГА
ТОР + ГИППОПОТАМ + 3.14
159—2.78128
```

Стоит употребить немного здравого смысла, чтобы определить такой способ переноса длинных операторов или выражений, который обеспечивает большую их читабельность.

4. Хотя это и не очень важно, часто программа легче читается, если символы арифметических операций *(+, —, /,* и т. д.) предваряются и сопровождаются одним или большим числом пробелов (при этом следует придерживаться постоянного числа таких пробелов).

5. Программируя на языках, допускающих запись операторов в «произвольном формате», старайтесь располагать начало записи каждого оператора в одном и том же столбце листинга программы. На языке ассемблера каждый компонент команды — код операции, поля регистров, поля адресов операндов, комментарии и т. д.— следует начинать с одних и тех же позиций. Как мы уже заметили в разд. 5.2.2, операторы IF-THEN следует располагать ступенчато, если употребляются вложенныеструктуры; анадогичные и основанные на здравом смысле правила следует использовать в применении к DO-циклам в Фортране, блокам BEGIN-END в Алголе и ПЛ/1 и. т. д.

6. Встречается много случаев, когда имеет смысл организовать метки операторов, имена процедур, имена абзацев и т. д. в форме возрастающей последовательности. Так, в Фортране разумно употреблять номера операторов (т. е. номера, которые обычно пишутся в колонках 1—7 исходной программы) в возрастающей последовательности; многие программисты на Коболе записывают имена абзацев в форме ПРОГРАММА-НАЧИСЛЕНИЯ-НАЛОГА-0010, ПОДПРОГРАММА-ВЫХОДА-0020 и т. д., с тем чтобы читатель знал, где в листинге программы искать любую метку (и связанный с нею программный код). Существуют возможности различных вариаций этого правила (как, например, приписывание операторам FORMAT в Фортране последовательных номеров специального множества), однако следует принять некоторый разумный набор соглашений и постоянно придерживаться его.

7. Снабжайте листинг программы подробным комментарием. Важность комментария уже обсуждалась в гл.1; здесь нас интересует только организация комментариев. Лично я всегда верил в лозунг, выдвинутый Крайтцбергом и Шнейдерманом в *The Elements of FORTRAN Style,* Harcourt, Brace, Jovanovich, New York, 1971. «Всегда документируйте программу подробнее, чем вам кажется необходимым». Однако многие программисты и слушатели, которые читали листинги моих программ (как на практике.такив этой книге), жаловались на то, что в них содержится слишком много комментариев и что операторы программы маскируются комментариями. Хотя по этому вопросу часто возникают горячие споры, следующие положения как будто всегда принимаются единодушно:

а) Перед каждой подпрограммой, процедурой или модулем *чрезвычайно* полезно помещать «пролог». Это словесное описание должно содержать краткое описание назначения модуля, способ обращения, характер действий и выходных данных, перечисление других подпрограмм, требуемых для его исполнения, других переменных и регистров, которые используются, разрушаются или как-то зависят от работы модуля, имя автора, дату создания первоначальной версии и дату последней модификации, а также краткое описание того, *как* модуль работает.

б) Содержательный комментарий должен сопровождать наиболее важные разделы программы, однако они не должны отвлекать внимание читающего от самого программного кода. Наиболее просто, конечно, это достигается в языке ассемблера; в языках высокого уровня эта задача обычно решается соответствующей организацией программы. Нужно, чтобы читатель имел возможность опускать комментарий, если он ему не доверяет или предпочитает работать без комментариев.

5.2.4. Избегайте сложных команд в языке ассемблера

Как мы уже отметили, язык ассемблера в принципе представляет собой наиболее опасный язык программирования, что объясняется огромным числом вариантов, которые программист может использовать в качестве решения его задачи на этом языке. Хотя соответствующие предосторожности могут быть несколько иными в разных машинах, следует назвать несколько таких, которые имеют общий характер:

1. Многие машины пословной адресации могут управляться командами, позволяющими обходить следующую команду, если в сумматоре содержится положительное число, отрицательное число, нуль и т. д. В связи с этим получили распространение *вложенные* конструкции таких операторов обхода, которые совершенно недоступны пониманию кого бы то ни было, за исключением их автора, если *он* вообще понимает, что именно он делает! В качестве при-

мера рассмотрим следующий простой фрагмент программы на языке ассемблера

```
SKIPL   CAT     , ОБОЙТИ СЛЕДУЮЩУЮ КОМАНДУ ЕСЛИ CAT МЕНЬШЕ НУЛЯ
SKIPE   DOG     , ОБОЙТИ СЛЕДУЮЩУЮ КОМАНДУ ЕСЛИ DOG РАВНО НУЛЮ
SKIPN   MOUSE   , ОБОЙТИ СЛЕДУЮЩУЮ КОМАНДУ ЕСЛИ MOUSE ОТЛИЧНО ОТ НУЛЯ
SKIPG   BIRD    , ОБОЙТИ СЛЕДУЮЩУЮ КОМАНДУ ЕСЛИ BIRD БОЛЬШЕ НУЛЯ
```

2. В некоторых машинах язык ассемблера содержит команду «исполнить». Например, команда

```
EXE  SWITCH
```

могла бы вызвать исполнение команды, расположенной по метке SWITCH; после этого обычно управление к команде, следующейза командой EXE, если только команда, расположенная по метке SWITCH, не была командой передачи управления, вызова подпрограммы, обхода и т. д. В некоторых ситуациях команда EXE оказывается чрезвычайно полезной, поскольку является очень эффективной операцией «однокомандного обращения к подпрограмме». К сожалению, простоты ради в некоторых машинах допускается вложенное использование команды EXE, т. е. команда, расположенная по метке SWITCH, сама может быть некоторой операцией EXE, и этот процесс вложения можно продолжать неопределенно глубоко. Трудно найти много практических примеров, где бы такая возможность была полезной, к счастью, ею располагают не очень многие машины!

3. Существует также несколько машин, в которых допускается многоуровневая косвенная адресация, часто в дополнение к различным уровням пре- и постиндексации. Один уровень косвенной адресации является наиболее распространенным (за особым исключением Систем IBM/360 и IBM/370) и обычно отражает разумную практику программирования; два уровня встречаются редко (слава богу!),хотя можно назвать несколько случаев, когда эта возможность весьма полезна; системы с более чем двумя уровнями редко кому-либо интересны, кроме неугомонных любителей изобретения способов использования многоуровневой косвенной адресации. Правда, есть польза в многоуровневой индексации многомерных таблиц (как это было сделано в операционной системе машины GE-435, которая отличается одной из наиболее изощренных в мире схем адресации), однако существуют другие, чуть менее эффективные, но значительно более ясные способы решения той же задачи. Подчеркиваем еще раз, что должное применение всех этих рекомендаций предполагает обращение к здравому смыслу и соответствующий'анализ возможных

издержек; обычно за ясность приходится расплачиваться эффективностью. Если применяется сложный подход, то тщательное документирование *совершенно* необходимо.

5.2.5. Разные предложения

Для всякого конкретного языка программирования или конкретной машины можно было бы высказать ряд дополнительных «что делать и что не делать», которые будут способствовать созданию простых программ. Довольно грустно осознавать, что во многих больших организациях подобные наставления возведены в ранг *стандарта,* поскольку такая мера часто сводит на нет первоначальную идею. Наша цель в действительности состоит в том, чтобы определить некоторую позицию по отношению к разработке программ, которые могут читататься другими. Вместо того чтобы погружаться в подробное обсуждение каких-либо новых рекомендаций, мы лучше завершим эту главу простым перечислением предложений разного характера, которые помогают избежать «нечитабельности» программ, составленных на Фортране, Коболе и других языках высокого уровня.

1. Если это возможно, то избегайте отрицаний в булевых выражениях; представляется, что их понимание составляет трудность для многих программистов. Например, выражение

```
IF NOT FLAG THEN A = B ELSE X = Y;
```

можно было бы переписать в следующей форме: .

```
IF FLAG THEN X = Y ELSE A = B,
```

хотя возможны случаи, когда запись, удобная для чтения, приводит к чуть менее эффективной программе. Многие составные булевы выражения, включающие логическое отрицание, могут быть переписаны в эквивалентной «позитивной» форме, которую проще понять.

2. Избегайте без нужды использования сложных составных булевых выражений. Примеры этого можно часто встретить в Кобол-программах, когда программист пишет

```
IF A AND B OR NOT C THEN PERFORM X.
```

На самом деле Кобол характеризуется возможностью написания составных булевых выражений, смысл которых кажется неопределенным читателю и часто совершенно не совпадает с намерениями программиста. Даже если нет неоднозначностей, такие выражения обычно с трудом понимают все, за исключением их автора.

3. Избегайте беспорядочного использования оператора EQUIVALENCE в Фортране, оператора REDEFINE в Коболе и аналогичных операторов в других языках программирования — в том

числе и языке ассемблера. Если для обозначения одной и той же константы, переменной или таблицы было употреблено несколько имен, то читателю программы будеттрудно запомнить значения этих имен и причины такого переопределения (предполагаем, что причины были — часто к этому прибегают, компонуя исходную составную программу из нескольких различных модулей, с тем чтобы обеспечить согласование переменных).

4. Заслуживают порицания передачи управления внутрь цикла иливыходизциклавпроизвольных точках (например, в DO-циклах на Фортране, PERFORMVARING-циклах на Коболе и т. д.). Многие языки предусматривают защиту от подобных шуток со стороны программиста; программисты, упорно следующие такой практике, создают многие трудности тем, кто работает с их программами.

5. Избегайте случаев произвольного изменения значения счетчика цикла. В большинстве языков существуют конструкции, позволяющие наращивать (или уменьшать) N раз значение счетчика до тех пор, пока он не исчерпает все значения, после чего программа переходит к исполнению команды, расположенной непосредственно за телом цикла. Исследование программ на Фортране[1)] показало, что в 95% случаев DO-циклы характеризуются единичным отрицательным приращением счетчика цикла. Естественно ожидать, что то же справедливо и в отношении большинства других языков программирования. Поэтому крайне обескураживают ситуации, когда в программе встречается цикл, *нормально* исполняемый с единичным приращением счетчика, в котором иногда значение счетчика вдруг изменяется на 13 (неожиданное приращение обычно происходит где-то внутри тела цикла).Следует повторить еще раз, что в некоторых языках подобная практика не допускается (передачей начального значения счетчика в специальный регистр), однако во многих из них это достигается очень просто — к соблазну тех программистов, которые пытаются извлечь из этого какую-то пользу.

ЛИТЕРАТУРА

1. Kernighan B. W., Plauger P. J., ProgrammingStyle: Examples and Countereamples, ACM Computing Surveys, Dec. 1974, pp. 303—319.
2. Kernighan B. W., Plauger P. J., The Elements of Programming Style, McGraw-Hill, 1974.
3. Schneiderman B., Kjeitzberg C., The Elements of FORTRAN Style, Harcourt, Brace, Javanovich, 1971.
4. Weinberg Q. M., The Psychology of Computer Programming, Van Nostrand, Reinhold, 1971.
5. Yohe J. M., An Overview of Programming Practices, ACM Computing Surveys, Dec. 1974, p. 221—246.

1) Knuth, An Emperical Study of FORTRAN Programs, Software Practice and Experience, v. 1, № 2, p. 105—133.

1. Как часто другие программисты читают ваши программы? Является ли такая практика обычной в вашей организации? Если нет, то почему?

2. Почему важно, чтобы вашу программу могли прочесть другие программисты?

3. Почему структурное программирование облегчает понимание программы?

4. Обладает ли ваша вычислительная система какими-либо средствами многоцелевого применения? Используются, ли они в каких-нибудь прикладных задачах в вашей организации? Если да, то попытайтесь собрать статистические данные для определения того, требуется ли больше времени на отладку и усилий на сопровождение таких программ (или систем), чем на разработки, не использующие таких средств. Существуют ли такие типы разработок, в которых многоцелевой подход позволяет создавать более простые и легкие для понимания программы?

5. Приведите три примера специальных приемов программирования на Коболе, которые можно отнести к разряду ухищрений. Для каждого такого приема укажите возможные виды ошибочной их интерпретации посторонним программистам. Считаете ли вы, что это может затруднить сопровождение программ?

6. Приведите три примера специальных приемов программирования на Фортране, которые можно отнести к разряду ухищрений. Для каждого такого приема укажите возможные виды ошибочной интерпретации программы другим программистам. Считаете ли вы, что это может затруднить сопровождение программы?

7. Приведите три примера специальных приемов программирования на ПЛ/1, которые можно отнести к разряду ухищрений. Для каждого такого приема укажите возможные виды ошибочной интерпретации программы другим программистам. Считаете ли вы, что это может затруднить сопровождение программы?

8. Приведите три примера специальных приемов программирования на языке ассемблера, которые можно отнести к разряду ухищрений. Для каждого такого приема укажите возможные виды ошибочной интерпретации программы другим программистам. Считаете ли вы, что это может затруднить сопровождение программы?

9. Приведите пример программы, модифицирующейся во время исполнения. Сколько времени ЦП экономится при таком подходе в сравнении с вариантом программы, *не* использующей модификацию? Выразите эту величину в процентах к полному времени исполнения программы, т. е., например, экономит ли применение оператора ALTER в Кобол-программе 1% полного времени ее исполнения? Имеет ли это большое значение? Оправдывает ли это использование модифицируемых программ?

10. Почему запись вычисления вида A=B*C+3**D/—3*X+Y*4 трудна для понимания? Сколькими компонентами [операторами и (или) операндами] следовало бы ограничить запись одного вычисляемого выражения?

11. Достигается ли применением скобок большая читабельность сложного арифметического выражения? Например, считаете ли вы, что выражение

$$A = (B * C) + (((3 ** D)/(-3)) - * X) + (Y * 4)$$

прочесть проще, чем выражение из п. 10? Можете ли вы предложить простой метод проверки правила парности скобок (т. е. требования, чтобы каждой правой скобке отвечала одна левая скобка), входящих в данное выражение? Сколько уровней скобок, на ваш взгляд, следовало бы допускать в записях арифметических выражений?

12. Почему, по вашему мнению, во многих организациях запрещено пользоваться в Кобол-программах вложенными условными операторами? Какие из приводимых ниже доводов (выдвинутых опытными программистами-слушателями нескольких курсов программирования, руководимых автором) являются наиболее важными?

а) «Вложенные условные операторы транслируются компилятором в ошибочный объектный код».

б) «Вложенные условные операторы транслируются компилятором в неэффективную машинную программу».

в) «Я не могу разобраться во вложенных условных операторах».

г) «Руководством по Коболу не рекомендуется пользоваться вложенными условными операторами».

д) «В синтаксисе Кобола не всегда просто определить однозначное соответствие между предложениями ELSE и условиями IF».

е) «В многоуровневые вложенные условные операторы трудно вносить новые предложения или стирать старые».

ж) «Никто никогда не учил меня пользоваться вложенными условными операторами».

13. Было давно отмечено, что программирующие на Алголе и ПЛ/1 (среди прочих) обычно более охотно пользуются вложенными условными операторами, чем программирующие на Коболе. Согласны ли вы с этим? Считаете ли вы, что следствием этого являются большие трудности в понимании и сопровождении программ на Алголе и ПЛ/1 в сравнении с программами на Коболе?

14. Часто высказывалось предположение, что вложенные условные операторы легче понимать, если они имеют соответствующую ступенчатую организацию (или оформление). Насколько, по-вашему, это важно?

15. Имеются ли в вашей организации какие-нибудь стандарты на число операторов, которые могут быть записаны в одной строке листинга? Приняты ли эти стандарты как обязательные для всех? Считаете ли вы, что это важная мера?

16. Ряд других предложений по оформлению программ перечислен в разд. 5.2.3. Приняты ли они официально в вашей организации? Считаете ли вы, что это имеет большое значение?

17. Многие из предложений, перечисленных в разд. 5.2.3, реализованы в составе некоторых языков программирования, например пакет программ TID4, поставляемых некоторыми разработчиками в составе Фортрана, пакет PRETTYPRINT в составе языка Лисп, пакет NEATER2 в ПЛ/1, пакет Метакобол и т. д. Имеется ли возможность пользоваться какими-либо из этих автоматизированных средств в вашей организации? Используются ли они? Считаете ли вы, что эти средства оказывают существенную помощь? Если нет, то почему?

18. В разд. 5.2.4 рекомендуется избегать применения в языке ассемблера вложенных операторов передачи управления. Считаете ли вы, что это важно? Можете ли вы представить такую ситуацию в программировании, когда такой подход требуется по существу? Можете ли вы предложить другой способ записи последовательности вложенных передач управления (такой, например, как рассмотренный в разд. 5.2.4), который был бы проще для понимания?

19. Позволяет ли язык ассемблера вашей машины пользоваться вложенными командами EXE? Если да, то приведите пример, в котором такой способ программирования был бы полезным. Если бы использование вложенных команд EXE было запрещено (из соображений простоты), то какими альтернативными методами программирования можно было бы воспользоваться.

20. Допускается ли вашей машиной многоуровневая косвенная адресация? Допускаются ли различные сочетания косвенной адресации и индексирования (например, преиндексирование и постиндексирование)?

ГЛАВА 6

ПРОГРАММИРОВАНИЕ С ЗАЩИТОЙ ОТ ОШИБОК

6.0. Введение

В данной главе рассматривается программирование с защитой от ошибок, которое я склонен иногда называть «прот.ивоошибочным». Защита от ошибок — это методология написания программ, уменьшающая вероятность ошибок и позволяющая сделать их более явными для программиста и для пользователя в тех случаях, когда они возникают (что неизбежно). Другими словами, мы предполагаем, чтов программуобязательно вкрадываются какие-то ошибки, пока не будет доказано, что это не так; исходя из этого принципа «презумпции виновности», мы включаем в программу как можно больше средств контроля ошибок.

Такой подход к программированию имеет только одну цель: *бьютрее обнаруживать ошибки.* Как многократно указывалось в нескольких предыдущих главах.тестированиеиотладкаотносятсяк наиболее трудоемким видам работ в программировании вычислительных машин. iВ гл. 7 и 8 мы рассмотрим ряд эффективных методов, помогающих решить эти трудные проблемы; однако все они обладают тем недостатком, что могут быть использованы только после того, *как программа уже написана.* Как будет показано в гл. 8, отладка чаще всего представляет собой чисто мыслительную работу. Хотя программные листинги (трассировочные таблицы) и распечатки содержимого оперативной памяти (дампы), безусловно, необходимы и полезны, многие из более тонких ошибок можно найти только путем длительных размышлений и кропотливого анализа обстоятельств, при которых возникает ошибка, и длительного тщательного анализа всех улик на основании «следов преступления», обнаруживаемых в различных распечатках и листингах. Особенно прискорбно наблюдать, когда программист (или, быть может, целая группа программистов, поскольку в большой системе зачастую неизвестно, чья подпрограмма вызывает ошибку) *вручную* производит поиск ошибки, в то время как эту ошибку при наличии средств контроля может легко обнаружить сама программа.

Все эти принципы разработки программ, в том числе программирование с защитой от ошибок,— только принципы. Мы не можем

доказать, что их применение позволяет получить лучшие или более эффективные программы; мы не можем сформулировать никаких четких алгоритмов для разработки программ, защищенных от ошибок. Все, что мы можем сделать,— это указать подход, дать рекомендации, определить методологию, обеспечивающую написание хороших программ. Принцип программированиясзащитой от ошибок — безусловно, один из наиболее важных элементов искусства разработки хороших программ. Однако многие программисты относятся к такому программированию резко отрицательно, что меня сильно удивило, когда я начал читать лекции и проводить семинары по этим вопросам. Поэтому мы начнем с изложения наиболее распространенных возражений против этого принципа и постараемся представить, как мы надеемся, веские контраргументы. Затем мы обсудим те стороны программы, которые должны определяться с учетом методологии программирования с защитой от ошибок. Наконец, мы опишем некоторые вспомогательные практические приемы программирования, полезные для решения этой проблемы.

6.1. Возражения против программирования с защитой от ошибок

Неоднократно читая курсы по новым проблемам программирования в целом ряде стран, я с удивлением обнаружил, что большинство программистов возражают против самой идеи контроля ошибок, причем некоторые довольно шумно и страстно. Выслушав множество таких возражений, я пришел к заключению, что все они сводятся, по существу, к трем формулировкам, каждая из которых рассматривается ниже.

6.1.1. Несправедливо заставлять программиста включать средства контроля ошибок в свою программу

Наиболее распространенное возражение против принципа защиты от ошибок, которому посвящена эта глава, связано не с дополнительными затратами времени центрального процессора или времени программиста, но скорее с понятием о справедливости. Позиция многих программистов заключается в следующем: «Я исхожу из предположения, что мою подпрограмму вызывают с помощью допустимых аргументов. Если какой-то болван дает моему модулю неправильные данные, то он и виноват в отказе системы. Не вешайте на меня работу по обнаружению чужих ошибок!» Несколько другой вариант протеста звучит так: «Мой шеф сказал, что я могу считать все данные, полученные моим модулем, совершенно «чистыми»,

полностью лишенными ошибок, почему же вы требуете, чтобы я осуществлял еще какой-то контроль ошибок? Вам не кажется, что было бы довольно нелепо, если каждый займется организацией собственного контроля ошибок?»

Хотя позиции «если что-то не идет, то это не моя вина» и «шеф сказал, что все будет в порядке» можно понять, это ни в коей мере не сокращает затраты времени на программирование. Может оказаться, что наиболее удобное место для обнаружения ошибки как раз в вашей подпрограмме, так почему бы не предусмотреть это в вашей программе на благо всем остальным? Если вы из упрямства откажетесь включить средства контроля в свою подпрограмму, то это может привести к *усложнению* ошибки: когда ваша подпрограмма получит неправильные данные, она выдаст неправильные результаты, которые будут переданы следующей подпрограмме и т. д. Первоначальная ошибка, которая может быть очень небольшой и несущественной, начнет расти подобно злокачественной опухоли. После того как несколько подпрограмм обменяются между собой неправильными данными, массивы начнут портиться, базы данных непонятным образом окажутся «забитыми» неправильными записями и вся система в конце концов может развалиться.

Удивительно, что многие программисты горят подобным возмущением, когда им приходится включать дополнительные команды контроля в свой модуль (и тем самым, как им кажется, нарушать изящество его структуры), чтобы обнаруживать, по их мнению, «чужие ошибки». В то же время очевидно, что для *пользователя* любая ошибка есть ошибка, и его не интересует, кто виноват. Поэтому было бы естественным, если бы программисты объединили свои усилия, чтобы помочь друг другу в такой ситуации, однако они этого не делают.

6.1.2. Программирование с защитой от ошибок требует слишком больших дополнительных затрат времени центрального процессора и слишком большого объема памяти

Другое распространенное возражение состоит в том, что добавочные средства контроля ошибок, предлагаемые в этой главе, связаны с неприемлемо большими накладными расходами. «Если я буду контролировать *все* те места, где могли бы возникать ошибки,— часто говорят прграммисты,— то моя программа почти все машинное время будеттратить на контроль ошибок, а не на полезные вычисления». Это возражение может быть обоснованным, однако обычно оказывается, что разумный объем средств контроля ошибок практически вовсе не замедляет хода выполнения программы — зачастую всего

на несколько миллисекунд; это совсем невысокая цена, которую стоит заплатить за возможность выловить неожиданные ошибки. В том случае, когда контроль ошибок *все же* значительно замедляет ход программы, его всегда можно исключить после того, как программа отлажена; правда, я почти уверен, что большинство логических средств контроля *никогда* не будет исключено, поскольку большинство программ *никогда* не достигает состояния полной отлаженности.

Существует любопытный психологический довод в пользу включения в программу обширных средств контроляошибок уженачальной стадии разработки. В большинстве случаев программы, по-видимому, повинуются несколько перефразированному закону Паркинсона: они растут таким образом, что занимают (причем иногда даже несколько превышают) весь объем имеющейся памяти, а их быстродействие снижается до предела. В связи с этим в начале разработки, когда еще имеется некоторый запас по объему памяти и по загрузке центрального процессора, рекомендуется включать в программу полный набор средств контроля ошибок. Позднее, когда начнутся жалобы на большой размер и неэффективность программы и когда потребуется ее сократить и повысить ее быстродействие,— только тогда будет целесообразно исключить контроль ошибок! Помните: *гораздо легче изъять часть средств логического контроля ошибок после отладки программы, чем включать эти же средства, когда выяснится, что ваша программа не работает так, как ей положено.* Исключение операторов контроля ошибок требует обычно просто физического изъятия нескольких перфокарт из исходной колоды программы или установки режима условной трансляции (компиляции), при котором в новую версию программы операторы контроля ошибок не попадают. *Включение* новых операторов контроля ошибок, с другой стороны, может оказаться крайне болезненной процедурой, особенно если программа уже заняла весь предусмотренный для нее объем памяти!

Пожалуй, наиболее страстные возражения в связи с этим мне пришлось выслушать от группы программистов одного банка в Монреале. Они были убеждены, что методы контроля ошибок и программирования с защитой от ошибок, которые я предлагал, выйдут далеко за пределы возможностей их машины. Но к концу семинара, который я проводил, выяснилось, что центральный процессор их машины почти 80% своего времени не загружен, абольшие области памяти зачастую были не заняты и вполне могли бы использоваться более крупными программами. Учтите, пожалуйста, следующее:*подавляющеебольшинствовычислительныхсистемвмире работает с недогрузкой и легко могло бы выполнять дополнительные функции по контролю ошибок.*

6.1.3. Программирование с защитой от ошибок отнимает слишком много времени у программиста

Последнее распространенное возражение против программирования с защитой от ошибок связано с собственным временем программиста. Программисты опасаются, что они будут тратить слишком много времени на включение дополнительных операторов контроля ошибок в свою программу, в результате чего им не удастся никогда закончить свою работу. Действительно, если бы в программе пришлось контролировать *каждый* из возможных источников ошибок, то это значительно замедлило бы процесс программирования. Если, однако, подойти к решению этой проблемы здраво и оценить, какие ситуации *реально* требуют тех или иных средств контроля ошибок, то оно не займет слишком много дополнительного времени. Опять-таки следует иметь в виду, что гораздо легче включать некоторые дополнительные операторы контроля *в процессе написания программы,* чем возвращаться к программе, *после* того как она уже написана, пытаясь вставить подобные операторы. Когда вы пишете программу, все детали ,ее работы (и, быть может, слабые места) свежи в вашей памяти и вам потребуется совсем немного усилий, чтобы предусмотреть немного дополнительных операторов контроля. *После* того как программа уже написана, вы забудете некоторые детали, а включение любых операторов контроля потребует повторной трансляции или перекомпоновки всей программы, что само по себе является потенциальным источником ошибок!

Однажды мне пришлось столкнуться с полным неприятием этой идеи группой программистов, работавших в одном банке в Стокгольме. Трое из программистов были особенно раздражены — настолько, что даже не появились во второй день моего семинара по программированию. Несколько удивленный такой силой эмоций, я спросил их руководителя, почему на них так подействоваламоя лекция по программированию с защитой от ошибок. «О, сегодня они отсутствуют на ваших занятиях вовсе не поэтому,— сказал он,— среди ночи их вызвали, чтобы найти и исправить ошибку в системе Сберегательная касса, за работу которой они отвечают. По-видимому, при обработке данных за прошлый месяц они потеряли 3 000 счетов из главного файла, и это по случайному совпадению обнаружилось как раз вчера вечером!»

6.2. Что необходимо контролировать в программе?

Надеясь, что принцип программирования с защитой отошибокуже получил у вас некоторое признание, мы перейдем теперь к рассмотрению уязвимых мест программы. Хотя характер и объем операций контроля, которые целесообразно выполнять в программе, должны диктоваться здравым смыслом, общим руководством при введении контроля может служить приводимый ниже перечень элементов.

6.2.1. Входные данные от „внешнего мира"

Хотя это само собой разумеется, все же отметим, что большинство программ производит вычисления с использованием входных данных, поступающих извне, например с устройства ввода перфокарт, с накопителей на магнитной ленте или терминалов. В некоторых особых случаях входные данные могут формироваться автоматическими средствами (например, системами управления реальными процессами и системами сбора данных); однако чаще всего предполагается, что входные данные подготавливает *пользователь,* т. е. перфораторщица, руководитель, исполнитель-кодировщик или, быть может, специалист, разработавший программу.

Большинство программистов согласятся, что любые такие входные данные следует контролировать на достоверность. Необходимость такого контроля ошибок увеличивается, если пользователи не имеют достаточной квалификации и не знакомы с вычислительными машинами (или относятся к ним с недоверием), но контроль нельзя ослаблять и в случае, когда пользователь является автором программы. *Следует* предполагать, что от пользователя в вашу программу могут поступить ошибочные входные данные, даже если работа перфораторщицы проверяется дублированием этих же данных другой перфораторщицей. В одной организации считали, что такая верификация данных на перфораторе является достаточной гарантией контроля ошибок, пока не обнаружили, что перфораторщицы экономят время, пропуская вначале через контрольник чистые карты (чтобы получить таким путем небольшие вырезы сбоку карты как признак правильности данных), и только затем перфорировали требуемые данные на карте!

6.2.2. Обращения из других программ

В равной степени важно, чтобы программа контролировала аргументы и данные, сформированные внутри системы. Нельзя бездумно полагать, что к вашей подпрограмме будут обращаться, используя правильные аргументы; в случае ошибки в сопряженной подпрограмме вызов вашей подпрограммы может производиться и с использованием неправильного аргумента. Чем крупнее система, в которой работает ваш модуль, тем более настоятельным становится контроль ошибок такого типа.

Этот аспект программирования с защитой от ошибок вызывает, кажется, наиболее острую реакцию некоторых программистов. Они уже признают то, что «внешний мир» может направлять им неправильные входные данные; но мысль о том, что их коллеги-программисты могут делать то же самое, кажется им совершенно неприемлемой. Ответ на такое возражение в действительности совсем прост: даже программисты допускают ошибки.

6.2.3. Записи базы данных на магнитных дисках или лентах

База данных — еще один источник неприятностей для многих программистов; чем больше база данных, тем более вероятно постепенное накопление в ней ряда незначительных ошибок. Некоторые программы могут внести ошибку при обновлении базы данных; в процессе считывания из внешнего накопителя или записи на него данные могут быть искажены за счет невыявленных аппаратных дефектов; некоторые записи могут быть по непонятным причинам утеряны при переписи базы данных с диска на ленту; оператор может использовать несколько устаревшую версию базы данных, когда он начинает запускать программу.

В одной организации постоянно возникали серьезные неприятности такого рода, пока не обнаружилось, что ошибки в базе данных вносились одной или несколькими программами на Фортране, которые периодически обновляли этот файл. Каждая обновляемая запись содержала некоторые числовые поля и некоторые текстовые поля. Числовые поля, в частности, включали числа с плавающей точкой, записанные с помощью оператора FORMAT, примерно эквивалентного оператору PICTURE языка Кобол. Если, однако, программа пыталась выдать число, слишком большое для оператора FORMAT, то Фортран-подпрограммы ввода-вывода системы программирования выдавали поля звездочек *без какого-либо дополнительного указания ошибки.* Естественно поэтому, что, когда какая-то другая программа пыталась прочитать такую запись (а ввиду периодического характера обработки данных внутри системы это нередко происходило только полгода спустя), она обнаруживала звездочки вместо числа с плавающей точкой, и программа обычно выбрасывалась в середине трехчасового прогона!

Программиста, чей модуль считывал записи базы данных, попросили включить какие-либо средства контроля ошибок, поскольку не было известно, какая именно Фортран-программа приводила к порче файла. Программист возмутился: «Почему я должен контролировать эти звездочки? Я их туда не записывал. Я не виноват, что моя программа читает всякую чушь!» Затем он сказал, что тщательный контроль ошибок для всех числовых полей каждой из 200 тыс. записей базы данных привел бы к удвоению времени работы программы, что, по-видимому, соответствовало действительности. После длительных горячих дебатов, однако, было решено, что довольно простым выходом из положения может оказаться программа «диагностики базы данных». Эту программу начали выполнять еженедельно, что позволило выявлять в файле неправильные числовые поля, неправильные текстовые поля, «посторонние» записи, пропущенные записи и различные другие ошибки, которые были сочтены наиболее вероятными. Программа требует приблизительно 10 мин

на каждый прогон (причем почти все это время занимают обмены данными с внешними накопителями — объем вычислений очень невелик) и экономит несколько часов в месяц за счет исключения повторных прогонов рабочих программ, что позволяет избежать потери записей и недоуменных вопросов со стороны операторов ЭВМ и, кроме того, избавляет пользователей от излишних огорчений.

6.2.4. Оператор вычислительной машины

Многие программисты, по-видимому, считают оператора вычислительной машины как бы безошибочным устройством ввода-вывода. К сожалению, хотя операторы обычно лучше обучены, более ответственны, добросовестны и менее склонны ошибаться, чем прочие пользователи, и они все же иногда *допускают ошибки.* Это, кажется, еще в большей степени справедливо для сложных мультипрограммных систем, систем дистанционной пакетной обработки и систем с разделением времени, которые сегодня столь широко распространены. Некоторые операторские ошибки вызывают отказы *системы;* однако большинство подобных ошибок приводит к тому, что только одна программа выполняется с неправильными входными данными или выбрасывается. Именно такие ошибки и являются темой нашего обсуждения.

Существуют самые разнообразные ошибки, которые может допускать оператор ЭВМ, но все они, по-видимому, могут быть разделены на несколько укрупненных категорий:

1. Оператор может запустить неправильную версию вашей программы — либо устаревшую версию, которую только что заменили новой, либо, наоборот, новую, но пока еще не отработанную до конца версию. Хуже всего то, что он может назвать вам (в большинстве случаев без злого умысла) не ту версию программы, которую он запускал в действительности.

2. Он может предпринять неправильные управляющие действия при выполнении вашей программы. Например, он может ввести неправильные управляющие карты (если ввод карт входит в его обязанности); присвоить вашей программе приоритет, который окажется слишком высоким или слишком низким; выделить объем памяти, недостаточный для эффективной работы программы; выбросить программу, если ему покажется, что в ней произошло зацикливание, в то время как на самом деле она выполнялась правильно; не выбросить программу, когда в ней действительно произошло зацикливание, что приводит к потерям дорогостоящего машинного времени; забыть заменить бумагу (сменить формат бланков) в печатающем устройстве в соответствующий момент времени и т. д.

3. Оператор может ввести неправильные исходные данные для вашей программы. Большинство программ требует ввода данных с перфокарт, с магнитной ленты или диска, и в обязанности опера-

тора часто гходитзакладка колоды перфокарт в карточное устройство ввода, установка кассеты (кассет) магнитной ленты на накопитель и т. д. Если допустить здесь ошибку, то программа может быть выполнена полностью, но с устаревшей версией базы данных. Аналогично многие машинные программы иногда требуют, чтобы оператор ЭВМ указал определенные «исключительные» параметры, напримердату (еслиэти параметры нельзя получить от операционной системы), или сам принял решение о дальнейших действиях после обнаружения ошибки ввода-вывода. Еще раз отметим, что ошибка оператора может иметь серьезные последствия.

В качестве одного из примеров нелепых ситуаций, возникающих из-за ошибок оператора, рассмотрим следующую историю. Программист писал программу для считывания примерно тысячи перфокарт, *упорядоченных по номерам счетов клиентов;* эти перфокарты затем должны были использоваться для обновления записей главного файла, также упорядоченных по номерам счетов. Программист решил построить программу как двухшаговую. При первом прогоне программа считывала все карты и проводила полное редактирование (контролировала, что все поля на карте располагаются в правильной последовательности, согласно номерам счетов, и т. д.); при втором прогоне программа *вновь считывала те же самые карты* и теперь уже обрабатывала главный файл. Предполагалось, что второго прогона не будет, если при первом обнаружатся ошибки. Все шло хорошо в течение первых шести месяцев; затем число карт с изменениями начало медленно расти, по мере того как в файл добавлялись счета новых клиентов и по мере повышения активности клиентов. И однажды случилось то, что и должно было случиться. При первом прогоне программа прочитала перфокарты и нашла, что все в порядке. Оператор вынул колоду из устройства ввода, подготавливая второй прогон, *и уронил ее.* Считая, что ничего страшного не произошло, он поднял перфокарты, тщательно, как только мог, собрал их в колоду и запустил на второй прогон. При втором прогоне программа, «не подозревая», что ей могут быть представлены неправильные входные данные, прочитала карты, уложенные непоследовательно, обнаружила, что ей не удается найти нужную запись счета клиента в главном файле и сразу же вышла из решения; таким образом было потеряно *несколько часов* машинного времени! Наиболее любопытно в этой истории то, что оператор был строго наказан за свою оплошность, но никто даже ничего не сказал программисту!

Быть может, наиболее эффективный способ научить программиста писать программы с защитой от ошибок операторов ЭВМ — это заставить его в течение некоторого времени поработать в роли оператора. Мой опыт говорит о том, что многие из самых старательных, основательных и добросовестных программистов начинали свою карьеру как операторы ЭВМ. Я даже знаю одну или две организации,

в которой программисты периодически переводятся в вычислительный центр, где месяц работают в качестве начинающих операторов. Не имея такой базы, программист должен по крайней мере попытаться учитывать следующие рекомендации:

1. Необходимо как можно шире использовать автоматические средства контроля меток файла, предусмотренные в вашей операционной системе; это обязательно для работы с файлами, хранящимися на диске или на ленте (просто удивительно наблюдать, как многие программисты пытаются избежать этого,— обычно по лености), но может оказаться столь же важным, чтобы гарантировать выполнение правильной версии программы. Сама программа обычно также является файлом библиотеки — и важно, чтобы оператор не мог ввести в решение устаревшую версию или экспериментальную версию из личной библиотеки программиста.

2. Необходимо стремиться сделать программу как можно более независимой от оператора ЭВМ. Не нужно просить оператора вводить дату с машинного пульта, если вы можете получить эту информацию от операционной системы; не нужно спрашивать его, не производится ли прогон данной программы еженедельно, если вы можете каким-то образом определить это сами; не нужнопроситьего принимать решение относительно мер по восстановлению после ошибки, если вам удастся разработать полностью автоматическую заранее определенную процедуру, поскольку в случае возникновения ошибки оператор, вероятнее всего, будет как бы в стрессовом состоянии, пытаясь исправить положение, и при этом даже более склонен ошибаться, чем обычно.

3. Никогда не нужно полагаться на то, что оператор делает все правильно. Если он вводит для вашей программы входные данные с пультовой пишущей машинки или устанавливает кассету с лентой или колоду перфокарт, делайте двойной контроль, чтобы удостовериться в их правильности.

4. Необходимо тщательно продумывать последовательность действий оператора, требуемых для работы с вашей программой. Не просите ли вы оператора каждые пять минут менять формы бланков в печатающем устройстве? Не заставляете ли вы его мчаться от печатающего устройства к накопителям на магнитной ленте, а оттуда — к устройству ввода перфокарт, чтобы успеть выполнить требования вашей программы? Если это так, попытайтесь перестроить вашу программу таким образом, чтобы оператор мог эффективно использовать свое время и способности. Вы обнаружите, что это приведет к уменьшению числа ошибок и к тому, что оператор будет в большей степени удовлетворен своей работой.

5. Очень полезно регулярно угощать своего оператора пивом или чашечкой кофе и просить его советов по усовершенствованию вашей программы. Он может дать вам полезные предложения и

квадратные корни или логарифмы, должны проверять, что они не работают с отрицательными аргументами; подпрограммы, вычисляющие e^x, будут обычно работать правильно только в случае, если величина x меньше 88. Такой подход может быть принят для большинства научных приложений: для численного анализа, статистической обработки данных, решения дифференциальных и интегральных уравнений и т. д. Если, например, вы разрабатываете программу для выполнения каких-то длительных вычислений по обработке данных эксперимента, вы можете *знать* по прошлому опыту, интуитивно или из теории, что определенный параметр не должен выходить за некоторые пределы. Поэтому, по-видимому, полезно проводить соответствующий контроль в конце каждой итерации.

6.3.2. Программы для экономических расчетов должны контролировать диапазоны чисел и типы данных

Большинство программ для экономических расчетов должно оперировать данными различных *типов,* т. е. алфавитно-цифровыми данными, целыми числами (в двоичной, восьмеричной, десятичной или шестнадцатеричной системах счисления) и числами с плавающей точкой. В связи с этимчасто необходимо, чтобы программа для экономических расчетов могла определить, что представленные ей данные относятся к соответствующему типу. Когда программа считывает имя клиента с входной перфокарты, она должна удостовериться, что это имя не содержит посторонних включений в виде цифровых символов. Если же она считывает последовательность символов, предположительно цифровых, она должна убедиться, что это действительно так. Когда программа считывает запись базы данных, она должна контролировать, что ни одно из полей не содержит неправильных данных.

Программа для экономических расчетов могла бы также контролировать диапазоны различных аргументов. Программа составления платежных ведомостей могла бы контролировать, что вычисленные налоги не превышают для каждого служащего рассчитанной заработной платы. Программа регистрации заказов могла бы проверять, не заказано ли отрицательное количество товаров. Программа переучета товаров могла бы также проверять, нет ли в описи отрицательных чисел. Программа подтверждения платежеспособности при решении вопроса о возможности принятия счета к оплате могла бы контролировать, не превышает ли сумма платежа некоторую указанную величину. Система-картотека личного состава могла бы проверять, что никто из служащих не указал в качестве года своего рождения *текущий* год.

Значительная часть программ для экономических расчетов пишется скорее всего на языках Кобол и ПЛ/1; к счастью, эти языки обладают рядом диагностических возможностей, помогающих про-

граммисту обнаруживать определенные виды ошибок. Как правило, программист может обнаружить попытку деления на нуль, переполнение сумматора и потерю значимости, ошибки «диапазона» и т. д. Наиболее употребительные операторы контроля ошибок в стандартной версии языка Кобол перечислены в табл. 6.1, а аналогичные операторы языка ПЛ/1 — в табл. 6.2.

Для программ бухгалтерского учета или для любых программ, ориентированных на экономические расчеты и имеющих дело с большими объемами входных данных и большими файлами, обязательны, как правило, дополнительные средства контроля. Когда такая программа обрабатывает входные данные, характеризующие выполненные операции (по изменениям текущих показателей с перфокарт, магнитной ленты или, быть может, даже терминалов), она должна вести счет числу таких операций, сведения по которым она приняла и обработала. Для финансовых приложений может быть также обязательным подсчитывать итоговые суммы прихода и расхода денежных средств и т. д. Если, например, программа последовательно обрабатывает какие-либо входные файлы (скажем, главный файл при обсчете платежных ведомостей или в банковской системе), она должна следить за числом уже прочитанных записей. Эти подсчитанные программой текущие итоговые показатели следует периодически сравнивать с показателями, вычисленными вручную, просто для того, чтобы каждый мог видеть, что оператор не ронял колоду перфокарт при установке ее на устройство ввода и что накопитель на магнитной ленте и (или) операционная система не пропустили ни одной записи с исходными данными при обработке.

Для финансовых приложений требования по контролю ошибок могут быть довольно жесткими. Зачастую они диктуются правилами

Таблица 6.1

ОПЕРАТОРЫ КОНТРОЛЯ ОШИБОК В СТАНДАРТНОМ КОБОЛЕ

Ошибки ввода-вывода	Имя-файла
USE AFTER STANDARD ERROR PROCEDURE ON [1]	INPUT OUTPUT I/O
Арифметические ошибки Повелительный-оператор ON SIZE ERROR [2]	

1) ИСПОЛЬЗОВАТЬ ПОСЛЕ СТАНДАРТНОЙ ПРОЦЕДУРЫ ОШИБКИ ПРИ...
2) ПРИ ОШИБКЕ РАЗМЕРА

Таблица 6.2

ОПЕРАТОРЫ КОНТРОЛЯ ОШИБОК В ЯЗЫКЕ ПЛ/1

Оператор	Значение
ON CONVERSION	Ошибки в данных, подлежащих преобразованию при вводе или выводе
ON SIZE	Попытка записать число, превышающее допустимый размер переменной
ON FIXEDOVERFLOW	Переполнение при арифметических вычислениях с фиксированной точкой
ON OVERFLOW	Переполньние при арифметических вычислениях с плавающей точкой
ON UNDERFLOW	Потеря значимости при арифметических вычислениях с плавающей точкой
ON ZERODIVIDS	Попытка деления на нуль
ON SUBSCRIPTRANGE	Попытка обратиться к элементу, индекс которого лежит вне границ массива
ON NAME	Попытка прочитать не определенную переменную с помощью оператора GET DATA
ON UNDEFINEDFILE	Попытка открыть файл, который не определен
ON TRANSMIT (имя-файла)	Постоянные ошибки передачи при выполнении ввода-вывода (имя-файла)
ON RECORD (имя-файла)	Длина записи больше или меньше, чем ожидалось
ON KEY (имя-файла)	Непредусмотренный ключ при обращении к запоминающему устройству произвольного доступа
ON ENDPAGE	Попытка отпечатать на странице большее число строк, чем указано константой PAGESIZE
ON ENDFILE	Попытка чтения после того, как прочитана последняя запись файла

и требованиями ведения бухгалтерского учета в данной организации и указываются программисту как часть технического задания. Но в любом случае программист не должен снимать с себя ответственности за контроль: в его программе могут возникать такие ошибки, которые никогда даже не снились ревизорам и бухгалтерам!

6.3.3. Тщательно обдумывайте возможные ошибки ввода-вывода

Одна из наиболее трудных задач в программировании — задача контроля и исправления ошибок ввода-вывода. Хотя мы можем предложить некоторые общие рекомендации по этому вопросу, каждую ситуацию, по-видимому, необходимо анализировать отдельно. В большинстве случаев можно различать ошибки, *связанные с устройством,* и ошибки, *связанные с передачей данных.* При-

мером ошибки, связанной с устройством, является попытка программиста использовать внешнее устройство (допустим, накопитель на магнитной ленте или устройство ввода перфокарт), которое отсутствует, находится в состоянии неготовности или занято кем-либо другим. Ошибка устройства может возникнуть также в процессе выполнения программы: оператор ЭВМ может случайно нажать кнопку выключения питания накопителя на магнитной ленте в то время, когда с ним производится обмен данными, или жев накопителе на магнитныхдисках может произойти касание головкой поверхности диска в процессе чтения или записи данных.

Ошибки, связанные с передачей данных, значительно более распространены и обычно значительно менее серьезны. Быть может, самая распространенная из всех ошибок подобного рода — это *ошибка четности;* с помощью контроля четности центральный процессор, канал ввода-вывода или контроллер ввода-вывода обнаруживают неудачную попытку прочитать или записать данные. В некоторых внешних устройствах могут встречаться также различные виды *временных ошибок,* обнаруживаемых аппаратным способом. К той же самой категории мы обычно относим *запрещенные команды ввода-вывода:* попытку сделать перемотку для устройства ввода перфокарт, попытку вывести двоичные данные на быстродействующее алфавитно-цифровое печатающее устройство, попытку записать данные на накопители на магнитной ленте или магнитных дисках с заблокированной записью и т. д.

Четкое разделение ошибок устройства и ошибок передачи данных часто важно в плане дальнейших действий. Программисту, как правило, предоставляются средства для *восстановления* процесса счета после обнаружения ошибки передачи, но, когда происходит ошибка устройства, он обычно оказывается во власти операционной системы (которая в свою очередь часто зависит от аппаратуры!). Большинство языков программирования высокого уровня, например, дают программисту возможность обнаруживать конец файла; к сожалению, многие версии языка Кобол *не* позволяют программисту управлять восстановлением после ошибок четности. В то же время некоторые языки требуют, чтобы программист осуществлял обработку ошибки собственными силами, но только после того, как система выполнит свои подпрограммы контроля ошибок; это иллюстрируется примером программы на Коболе (рис. 6.1). И напротив, программист, работающий на языке ассемблера, обычно имеет достаточно широкие возможности управления обработкой ошибок.

Обработка ошибок ввода-вывода — это проблема, которая заслуживает тщательного изучения, когда вы начинаете писать программу. Для каждого из устройств ввода и устройств вывода составьте перечень возможных ошибок с указанием их вероятности; затем спросите себя, каким образом онимогут повлиять на ход вашей программы. Наконец, проанализируйте средства контроля

```
РАЗДЕЛ ПРОЦЕДУР;
ОБЪЯВЛЕНИЯ:
СЕКЦИЯ ОШИБКИ-ГЛАВНОГО-ФАЙЛА.
    ИСПОЛЬЗОВАТЬ ПОСЛЕ СТАНДАРТНОЙ ПРОЦЕДУРЫ ОШИБКИ
    ГЛАВНОГО ФАЙЛА
ПОДПРОГРАММА-ОШИБКИ-ГЛАВНОГО.
    ВЫДАТЬ СООБЩЕНИЕ-ОШИБКИ-ГЛАВНОГО НА ПУЛЬТ,
    ОСТАНОВ ПРОГРАММЫ.
СЕКЦИЯ ОШИБКИ-ФАЙЛА-ИЗМЕНЕНИЙ.
    ИСПОЛЬЗОВАТЬ ПОСЛЕ СТАНДАРТНОЙ ПРОЦЕДУРЫ
    ОШИБКИ ДЛЯ ФАЙЛА-ИЗМЕНЕНИЙ.
ПОДПРОГРАММА-ОШИБКИ-ИЗМЕНЕНИЙ
    ПЕРЕМЕСТИТЬ НОМЕР-ЗАПИСИ-ИЗМЕНЕНИЙ В НОМЕР-ЗАПИСИ-
    ОШИБКИ.
    ВЫДАТЬ СООБЩЕНИЕ-ОШИБКИ-ИЗМЕНЕНИЯ НА ПУЛЬТ.
СЕКЦИЯ ОШИБКИ-ВЫВОДА.
    ИСПОЛЬЗОВАТЬ ПОСЛЕ СТАНДАРТНОЙ ПРОЦЕДУРЫ ОШИБКИ
    ПРИ ВЫВОДЕ.
ПОДПРОГРАММА-ОШИБКИ-ВЫВОДА.
    ВЫДАТЬ СООБЩЕНИЕ-ОШИБКИ-ВЫВОДА НА ПУЛЬТ.
    ПРИНЯТЬ ОТВЕТ С ПУЛЬТА.
    ЕСЛИ ОТВЕТ РАВЕН НЕТ ТО ОСТАНОВ ПРОГРАММЫ.
КОНЕЦ ОБЪЯВЛЕНИЙ.
```

Примечание: На некоторых машинах имеются нестандартные варианты этих операторов, которые позволяют программисту определить действительный характер ошибки ввода-вывода. Для Системы IBM/360 рекомендуем читателю обратиться к IBM System/360 Operating System: American National Standard COBOL Programmer's Guide, Form GC 28-6399.

Рис. 6.1. Обработка ошибок ввода-вывода в Коболе.

ошибок, предусмотренные вашим языком программирования и операционной системой.и представьте себе, в достаточной ли степени вы можете управлять обработкой ошибок ввода-вывода.

6.3.4. Составляйте блок-схему и комментарии для своей программы, когда вы ее пишете, а не откладывайте это на „потом"

Значение хорошего документирования программ просто невозможно переоценить. Хотя основную ответственность за плохую документацию следует возложить на программиста, как правило, есть в чем упрекнуть и его руководителя. Документацию часто вообще опускают или сводят к минимуму в «сверхсрочных» случаях, что скорее всего означает, что руководитель неправильно оценил трудоемкость проекта. Аналогично руководство может оказаться вынуж-

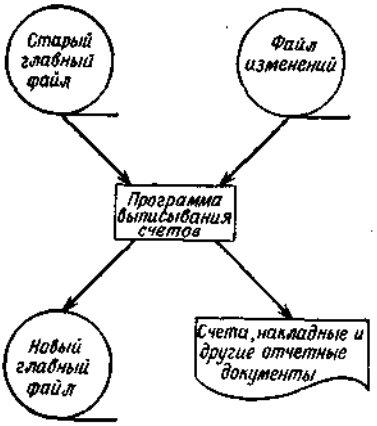
денным перевести программиста на другой проект, а в этом случае документация, по-видимому, никогда не будет сделана. Наконец, руководитель может позволить убедить себя правдоподобными искусными доводами, которые выдвигает программист, пытаясь избежать скучной работы по выпуску документации.

Поскольку в этой книге мы рассматриваем в основном вопросы программирования, мы не будем более глубоко вдаваться в многочисленные причины отсутствия хорошей документации, тем более, что этой темы мы уже касались в гл. 1 и 5. Следует, однако, подчеркнуть, что *хорошая документация может быть столь же полезной в процессе разработки и внедрения программы, как и впоследствии*. Сама работа по оформлению документации обычно заставляет программиста пересматривать основные моменты работы своей программы. *Почему,* собственно, приращение переменной XYZ производится в этой точке программы? Какие функции выполняет эта подпрограмма и какого рода параметры ей требуются? Программист не может не задать себе подобные вопросы, когда он пытается описать свою программу на естественном английском языке; и уже это часто позволяет ему выявить в своей программе различные дефекты, слабости, несоответствия и излишества.

6.3.5. Старайтесь избегать программ с большим числом комбинаций флажков

На всем протяжении этой главы мы внушали мысль о том, что программирование с защитой от ошибок — хороший путь обнаруживать ошибки тогда, когда они возникают. В некоторых случаях ошибки являются следствием того, что пользователь делает нечто, не предусмотренное программистом, например создает определенные комбинации входных данных, неожиданные для программиста. Аналогичная ситуация может складываться также и *внутри* программы: одна подпрограмма, или модуль, может делать что-то, к чему другая подпрограмма совсем не подготовлена. Особенно много хлопот может доставить программа, в которой внутренние условия и режимы контролируются с помощью многочисленных флажков, или программных переключателей.

Распространенную ситуацию иллюстрируетрис. 6.26. Программист имеет семь различных флажков для индикации различных событий, которые происходят в процессе обработки исходных данных об изменениях показателей (входного файла изменений). В различные моменты времени некоторые из модулей программы будут устанавливать (или сбрасывать) те или иные флажки; другие модули будут опрашивать (анализировать) флажки, чтобы определить, какие действия им следует предпринять. Профлема большого числа комбинаций флажков заключается в том, что программист редко представляет *полный* перечень возможных состояний программы.



Эта программа выполняет экономические расчеты и выписывает счета для крупной организации. При каждом прогоне программа считывает ГЛАВНЫЙ файл, содержащий данные обо всех старых клиентах (покупателях); она считывает также файл ИЗМЕНЕНИЙ, содержащий сведения о новых клиентах, заказы которых были получены уже после последнего прогона. Оба файла упорядочены соответственно по кодовым номерам клиентов, присвоенным им системой социального обеспечения. При работе программа выписывания счетов производит слияние обоих файлов, формируя новый ГЛАВНЫЙ файл как часть результата прогона.

Рис. 6.2а. Фортран-программа с использованием флажков.

На рис. 6.2б например, мы имеем семь флажков; предполагая, что любой флажок может иметь нулевое или единичное значение, получаем всего 128 различных возможных комбинаций этих флажков, а при этом программист может использовать всего дюжину из них. И если в программе содержатся ошибки, то весьма вероятно, что в некоторый момент вычислений установится *неправильная* комби-

```
C
C   ФЛАЖОКА = 0   ЕСЛИ МЫ ЕЩЕ НЕ СЧИТАЛИ НИ ОДНОЙ ЗАПИСИ
C                 ИЗ ГЛАВНОГО ФАЙЛА
C           = 1   ЕСЛИ МЫ УЖЕ СЧИТАЛИ ПО МЕНЬШЕЙ МЕРЕ
C                 ОДНУ ЗАПИСЬ ИЗ ГЛАВНОГО ФАЙЛА
C   ФЛАЖОКВ=-0    ЕСЛИ МЫ ОБРАБАТЫВАЕМДАННЫЕОСТАРОМ
                  КЛИЕНТЕ
C           =1    ЕСЛИ МЫ ОБРАБАТЫВАЕМ ДАННЫЕО НОВОМ
                  КЛИЕНТЕ
C   ФЛАЖОКС = 0   ЕСЛИ ЭТО НОВЫЙ КЛИЕНТ, КОТОРЫЙ РАНЕЕ НЕ
C                 БЫЛ В ЧИСЛЕ КЛИЕНТОВ
C           =1    ЕСЛИ ЭТО НОВЫЙ КЛИЕНТ, КОТОРЫЙ РАНЕЕ
C                 БЫЛ В ЧИСЛЕ КЛИЕНТОВ
C   ФЛАЖОКЕ = 0   ЕСЛИ ДЛЯ ЭТОГО КЛИЕНТА ЕЩЕ НЕ ВЫПОЛ-
                  НЕНЫ ВСЕ ВЫЧИСЛЕНИЯ
C           =1    ЕСЛИДЛЯЭТОГО КЛИЕНТАУЖЕ ВЫПОЛНЕНЫ
                  ВСЕ ВЫЧИСЛЕНИЯ
C   ФЛАЖОКЕ = 0   ЕСЛИ ПОКА ЕЩЕ НЕ ВЫВЕДЕНА НИ ОДНА ВЫ-
                  ХОДНАЯ ЗАПИСЬ
C           =1    ЕСЛИУЖЕВЫВЕДЕНАХОТЯБЫОДНАВЫХОД-
                  НАЯ ЗАПИСЬ
C   ФЛАЖОКО = 0   ЕСЛИ УЖЕ НЕТ ВЫХОДНЫХ ЗАПИСЕЙ ДЛЯ
                  ВЫВОДА
C           =1    ЕСЛИ ЕЩЕ ЕСТЬ ВЫХОДНЫЕ ЗАПИСИ ДЛЯ
                  ВЫВОДА
```

Рис. 6.26. Фортран-программа с использованием флажков.

Состояние	ФлажокA	ФлажокB	ФлажокC	ФлажокD	ФлажокE	ФлажокF	ФлажокG	Можетперейти всостояние
1	0	1	1	0	0	0	1	3
2	0	1	1	0	0	1	1	4
3	0	1	1	0	1	0	1	1,4,5,9
4	0	1	1	0	1	1	1	2,5,6,10
5	0	1	1	1	0	0	1	7
6	0	1	1	1	0	1	1	8
7	0	1	1	1	1	o	1	1,5,8,9
8	0	1	1	1	1	1	1	1,5,10
9	1	0	0	0	0	0	1	11
10	1	0	0	0	0	1	1	13
11	1	0	0	0	1	0	1	9,12,13,14
12	1	0	0	0	1	1	0	*Стоп*
13	1	0	0	0	1	1	1	10,15,20
14	1	1	1	0	0	0	1	16
15	1	1	1	0	0	1	1	17,18
16	1	1	1	0	1	0	1	9,11,14,18,19
17	1	1	1	0	1	1	0	*Стоп*
18	1	1	1	0	1	1	1	10,15,20
19	1	1	1	1	0	0	1	21
20	1	1	1	1	0	1	1	23
21	1	1	1	1	1	0	1	9,19,22,23
22	1	1	1	1	1	1	0	*Стол*
23	1	1	1	1	1	1	1	10,15,20

Рис. 6.3. Таблица состояний для этой же Фортран-программы.

нация значений флажков. Поскольку программа анализирует каждый раз только один или два флажка, маловероятно, что ошибка будет обнаружена достаточно рано, чтобы можно было сказать, в чем ее причина.

Альтернативный подход заключается в том, чтобы каждую допустимую комбинацию условий в программе определить как *состояние;* при этом состояние 0 могло бы соответствовать ситуации, когда значение признака A, или флажок A=0, флажок B=0 и флажок C=0, а состояние 1 — ситуации, когда флажок A=0,

флажок B=0, а флажок C=1. Такой подход имеет два преимущества. Во-первых, он заставляет программиста знать все допустимые состояния (поэтому программист может определить, о чем указывалось выше, что определенные комбинации флажков являются недопустимыми); и, во-вторых, он заставляет его перечислить все допустимые *изменения состояний*. Так, в процессе выполнения программы может потребоваться переход от состояния 0 к состоянию 2, но переход от состояния 2 к состоянию 0 может быть бессмысленным и, следовательно, недопустимым. Такой подход иллюстрируется рис. 6.3; здесь исходная программа, флажки которой были показаны на рис. 6.26, реализована с помощью кода состояний.

Все управление программой можно было бы обеспечить с помощью двух переменных, OLD-STATE (старое состояние) и NEW-STATE (новое состояние) и многократного применения вычисляемого оператора GO TO (или оператора CASE или GO TO DEPENDING и т. д.). Каждому из состояний, перечисленных на рис. 6.3, мог бы соответствовать в программе некоторый модуль, выполняющий вычисления согласно предусмотренной для него комбинации флажков. Таким образом, если NEW-STATE-14, управление передается модулю 14 (которому, надо надеяться, будет присвоено имя, более соответствующее назначению!). Прежде чем модуль 14 начнет основные вычисления, он проверит OLD-STATE, чтобы удостовериться, что управление ему передано от допустимого модуля (согласно рис. 6.3, в состояние 14 можно перейти только от состояния 16 и состояния 11). Когда модуль 14 закончит свою работу, он за-

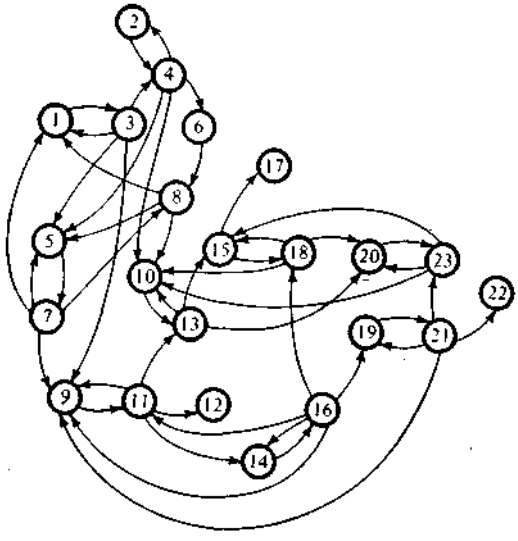


Рис. 6.4. Диаграмма состояний для Фортран-программы.

фиксирует текущее состояние в OLD-STATE, определит, каким должно быть следующее состояние (согласно рис. 6.3, состояние 14 может привести только к состоянию 16), и запишет код этого нового состояния в NEW-STATE. Затем осуществляется переход на управляющий вычисляемый оператор GO TO, который в свою очередь сразу же передаст управление следующему модулю.

Описанный способ состояний должен понравиться тем, кто освоил теорию переключательных схем и (или) теорию конечных автоматов. Мы рекомендуем, чтобы программист представлял свою программу как автомат с конечным числом состояний; разработка программы таким способом включает вычерчивание диаграммы состояний. Диаграмма состояний для нашего примера программы показана на рис. 6.4. Тем, кому эта терминология и диаграмма рис. 6.4 покажутся незнакомыми и непривлекательными, стоит, по-видимому, сказать, что этот способ может рассматриваться как один из вариантов *таблиц решений*. Основная мысль всех этих рассуждений заключается в том, что программы с большим числом бессмысленных комбинаций флажков весьма уязвимы и вряд ли смогут надежно работать.

6.3.6. При условных переходах проверяйте все возможные ситуации

Существует много ситуаций, когда программист анализирует переменную величину, которая, как предполагается, может иметь одно из двух значений — «истинное» или «ложное», «положительнсе» или «отрицательное» и т. д. Если анализ показывает, что значение переменной *не* является первым из двух возможных, программист, естественно, считает, что оно должно быть вторым, и предпринимает соответствующее действие. Опасность такого подхода можно продемонстрировать, рассказав о случае, который произошел с одним оператором ЭВМ. Оператор обнаружил, что программа напечатала на пультовом телетайпе бессмысленное сообщение, сопровождаемое вопросом:

ВЫ ЖЕЛАЕТЕ ПРОДОЛЖАТЬ?

Не зная, что делать, и не понимая, что значит предыдущее ошибочное сообщение, оператор в ответ напечатал:

ПОМОГИ

Между тем подпрограмма ввода, составленная программистом, ожидала, само собой разумеется, ответа в виде одного знака «Y» (да) или «N» (нет). Вначале она проверила, не является ли входное сообщение сообщением «Y»; обнаружив, что не является, она передала управление подпрограмме «нет», которая немедленно остановила все вычисления и перешла на «конец задачи»!

Вывод, по-видимому, ясен: если вы анализируете переменную, которая может иметь N известных значений, включайте в программу (N+1)-ю контрольную проверку, чтобы убедиться, что нет ошибки в виде значения, отличного от всех предусмотренных.

6.3.7. Не допускайте открытого соединения модулей

Программисты, которые упорно продолжают писать неструктурированные программы с операторами безусловных передач управления, нередко обнаруживают, что модуль А, например, заканчивается переходом на начало модуля В (под «модулем» мы понимаем здесь секцию Кобола, группу операторов Фортран-программы и т. д.). Располагая модуль В непосредственно после модуля А в исходной программе, программист может осуществить открытое соединение модулей, т. е. сделать так, что модуль А будет переходить в модуль В, и благодаря этому сэкономить лишний оператор GO TO. Таким образом, мы можем представить себе следующую гипотетическую последовательность команд на любом языке программирования:

```
A:   ЗАГРУЗИТЬ X
     ПРИБАВИТЬ Y     ВЫЧИСЛИТЬ X+Y
     ЗАПОМНИТЬ Z     ЗАПОМНИТЬ РЕЗУЛЬТАТ В X,
                     ОТКРЫТОЕ СОЕДИНЕНИЕ С
                     МОДУЛЕМ В
B:   : ЗАГРУЗИТЬ P
     ПРИБАВИТЬ R     ВЫЧИСЛИТЬ P+R
     ЗАПОМНИТЬQ
```

Если, однако, модули А и В относительно четко разделены по функциям, то в процессе отладки или при последующем сопровождении проекта могут произойти следующие неприятные вещи. Между модулями А и В в исходную программу может оказаться необходимым включить новый модуль; может также потребоваться переместить модуль А и (или) В в какое-то другое место исходной программы (с мини-компьютерными программами это случается довольно часто); изменения в модуле А или В могут нарушить логику открытого соединения одного модуля с другим. Поэтому гораздо лучше предусмотреть в модуле А *явную* команду передачи управления модулю В — тем более, что оператор GO TO является относительно дешевым в смысле затрат объема памяти и времени центрального процессора. Еще лучше принять методику структурного программирования, изложенную в гл. 4, и отказаться от операторов GO TO вообще. Используйте вместо этого команды вызова процедур, операторы IF-THEN или простые периодические структуры.

6.3.8. Контролируйте индексацию массивов

Как мы покажем в гл. 8, одна из наиболее распространенных ошибэк программирования — это выход индексов за пределы массива; подобная ошибка может испортить команды или данные и приводит обычно к неработоспособности программы.

Многие языки программирования включают дополнительные тестовые средства, которые заставляют транслятор генерировать вспомогательный объектный модуль для контроля обращений к массиву в процессе выполнения программы. Если язык, на котором вы работаете, предусматривает такую возможность, целесообразно, по-видимому, использовать ее при первых нескольких прогонах вашей программы; если нет, то вам стоит, быть может, включить в программу собственный модуль контроля ошибок. Поскольку контроль ошибок такого типа обычно существенно замедляет выполнение программы, со временем его можно будет исключить.

6.3.9. Проверяйте вычисляемые операторы GO TO

Фортран, Коболиряддругих языков программированияимеютоператоры типа

```
GO TO A, B, C, ..., Z DEPENDING ON I.
```

Обычно предполагается, что переменная I — целое число (если нет, то, как правило, производится ее преобразование в целое число). При выполнении оператора GO TO DEPENDING ON ожидается, что I имеет значение в пределах между I и *n,* где *n* — число адресов в списке. Если в программе содержится ошибка, то переменная I, естественно, может не иметь правильного значения.

Действие, предпринимаемое в том случае, когда I имеет значение, выходящее за допустимые пределы, зависит от языка и от конкретной реализации транслятора. Если I будет иметь значение меньше 1, управление обычно будет передано по первому адресу списка (т. е. A в приведенном выше примере) или оператор GO TO DEPENDING ON будет игнорироваться (т. е. управление перейдет к следующему за ним оператору). Аналогично если I будет иметь значение, превышающее *n,* управление будет передано последнему адресу списка (т. е. Z в приведенном примере) или же оператор GO TO DEPENDING ON будет игнорироваться. В любом таком случае выполнение программы зачастую продолжается самым неожиданным образом.

Предлагаемое средство защиты очень простое. В зависимости от того, как именно действует транслятор в подобных случаях (например, игнорируется ли оператор GO TO DEPENDING ON, когда переменная I имеет недопустимые значения), можно предусмотреть соответствующие команды, обеспечивающие передачу управления

подпрограмме анализа ошибки, когда I выходит за пределы допустимого диапазона. Это требует обычно очень небольших накладных расходов, и такие команды следует, по-видимому, оставлять в программе на все время.

ВОПРОСЫ

1. Приведите краткое определение метода программирования с защитой от ошибок. Каковы его цели? На каком этапе разработки программы, по вашему мнению, применение этого метода может быть наиболее эффективным?

2. Согласны ли вы с возражением против программирования с защитой от ошибок, рассмотренным в подразд. 6.1.1, а именностем, чтосправедливо просить программиста включать в свою программу средства контроля, чтобы можно было находить ошибки соседней программы? Разделяют ли это мнение другие программисты вашей организации? Допускается ли, чтобы их мнения доминировали над другими мнениями, т. е. могут ли они избежать включения такого контроля ошибок, если достаточно энергично противятся этому?

3. Согласны ли вы с возражением против программирования с защитой от ошибок, рассмотренным в подразд. 6.1.2, а именно с тем, что оно требует слишком больших затрат времени центрального процессора и объема памяти? Есть ли у вас данные количественной оценки накладных расходов, которые потребовались бы в случае достаточно полного контроля ошибок? Значительны ли эти затраты? Является ли это возражение общепринятым в вашей организации? Действительно ли ваша вычислительная машина настолько перегружена, что зто возражэние правомерно?

4. Если включение в программу дополнительных средств контроля ошибок *действительно* сопряжено с большими накладными расходами, можетели вы предложить какой-то путь решения этой проблемы?

5. Многие ассемблеры и компиляторы предусматривают возможность условного ассемблирования и условной компиляции. Например, программист может написать

```
%ERRORFLAG=1


%IF ERRORFLAG=1
    X=1
    Y = 2
    Z=3
%ENDIF
```

Отметим, что ERRORFLAG (ФЛАЖОКОШИБКИ) — переменная, которая существует только в процессе компиляции или ассемблирования; в объектной программе для нее места не выделено. Если программист хочет, чтобы в его программу при ассемблировании попал модуль контроля ошибок, он устанавливает ERRORFLAG равным 1; если он хочет исключить этот модуль, то может установить ERRORFLAG равным 0 и провести повторную компиляцию (ассемблирование) программы.

Альтернативный подход заключается в такой организации программы, при которой логика контроля ошибок *всегда* присутствовала бы в объектном коде. Считывая значение некоторой переменной (например, с пультовой пишущей машинки, как указание оператора или программиста), программа решает, нужно или нет выполнять модуль контроля ошибок. После такого вступления рассмотрите следующие вопросы:

а) Имеются ли в вашем языке программирования подобные возможности условного ассемблирования?

б) Используете ли вы их?

в) Применительно к средствам контроля ошибок является ли способ условного ассемблирования более эффективным, чем описанный выше альтернативный способ?

г) Какой из них более удобный?

д) Существуют ли какие-либо другие соображения и причины, по которым вы предпочли бы использовать условное ассемблирование, а не альтернативный способ включения — исключения средств контроля, или наоборот?

6. Приведите пример ситуации, где отсутствие контроля правильности внешних входных данных могло бы вызвать отказ в работе программы.

7. Насколько тщательно и всесторонне следует контролировать входные данные пользователя? Возможно ли представить себе ситуации, где контроль может оказаться *слишком* обременительным?

8. Считаете ли вы, что необходимо контролировать аргументы и параметры, поступающие от других программ или модулей, чтобы удостовериться в их правильности? Насколько серьезно и тщательно следует проводить такой контроль? Имеются ли при этом какие-либо неблагоприятные ситуации?

9. Считаете ли вы, что необходимо контролировать записи базы данных на магнитных дисках и лентах? Каким видам ошибок следует уделять особое внимание? Как часто они могут возникать? Имеются ли какие-либо неблагоприятные ситуации, связанные с контролем ошибок такого рода?

10. Насколько тщательно программисту следует контролировать возможные ошибки оператора ЭВМ? Считаете ли вы эту проблему важной?

11. Составьте перечень типов ошибок, которые мог бы сделать оператор ЭВМ,— ошибок, приводящих к возможному отказу или неправильному выполнению вашей программы. Для каждого типа ошибок укажите возможный способ контроля и защиты. Можно ли добиться в любом случае, чтобы оператор правильно выполнил вашу программу?

12. Как вы можете узнать, правильную ли версию вашей программы выполняет оператор? Требуются ли для этого какие-либо специальные средства в программе? Стоит ли это ваших усилий?

13. В подразд. 6.4.2 рекомендуется разрабатывать программу как можно более независимой отдействий оператора ЭВМ. Приведите три примера ситуаций, где программа сама могла бы принимать решение о дальнейших действиях, не обращаясь к оператору. Какой объем дополнительной работы по программированию потребовался бы для этого? Стоит ли это ваших усилий?

14. Считаете ли вы, что стоит тратить силы на включение в программу средств контроля ошибок, чтобы убедиться в правильности работы центрального процессора? Насколько это трудно, по вашему мнению? Можете ли вы представить себе какие-либо ситуации, для которых это было бы оправданным?

15. При каких условиях программисту следует включать в свою программу модуль контроля ошибок, чтобы убедиться в правильности работы системного программного обеспечения? Насколько тщательным должен быть такой контроль?

16. Приведите три примера числовых аргументов, требующих контроля в программах научных расчетов.

17. Приведите три примера контроля ошибок, необходимого при выполнении типичной программы экономических расчетов.

18. Считаете ли вы, что программист должен *всегда* предусматривать в своей программе контроль возможных арифметических ошибок, например деления на нуль, переполнения сумматора и т. д.? При каких условиях он может не беспокоиться о таком контроле?

19. Какой способ контроля ошибок следует принять при проверке правильности выполнения операций ввода-вывода? Предложите»некоторые общие принципы решения следующих проблем:

а) В программе, которая выборочно считывает записи из главного файла, подготавливая отчет для руководства, некоторые из записей не удается прочитать из-за ошибок четности.

б) В программе, которая выборочно считывает записи из главного файла, обновляет их и записывает в выходной главный файл, некоторые из записей главного файла не удается прочитать из-за ошибок четности.

в) В программе, которая распечатывает записи файла результатов, некоторые из записей не удается распечатать из-за ошибок четности.

20. Согласны ли вы с утверждением (см. разд. 6.3.4), что «хорошая документация может быть столь же полезной в процессе разработки и внедрения программы, как и впоследствии»? В чем, по вашему мнению, основное значение документирования программ?

21. Каким образом, по вашему мнению, следует стандартизовать комментарии к программам? Считаете ли вы, что целесообразно требовать комментария для каждого оператора исходной программы? Каким путем можно обеспечить, чтобы комментарии были четкими и вразумительными?

22. Согласны ли вы, что программа должна сопровождаться подробными блок-схемами? Как можно обеспечить, чтобы блок-схема точно представляла алгоритм, реализуемый программой в действительности?

23. Какие виды документации, по вашему мнению, необходимы программисту сопровождения для дальнейшего сопровождения написанной вами программы? Считаете ли вы, что в этой области могут быть разработаны какие-либо стандарты?

24. Какие опасные последствия может иметь наличие в программе многочисленных переключателей и флажков? Существуют ли какие-нибудь общие способы составления программ без флажков?

25. В порядке эксперимента найдите в своей организации программу, содержащую большое число флажков. Попытайтесь переработать эту программу, не применяя никаких флажков или переключателей; после этого ответьте на следующие вопросы:

а) Получилась ли новая программа более легкой в понимании и более удобной в сопровождении?

б) Не кажется ли вам, что новая программа будет иметь меньше скрытых ошибок, которые в конце концов придется устранять программисту сопровождения?

в) Не требует ли новая программа большего объема памяти или времени центрального процессора? Если да, то на сколько большего?

ПРИНЦИПЫ ТЕСТИРОВАНИЯ ПРОГРАММ[1)]

Для крупных систем коллективного пользования и систем реального времени затраты на тестирование изменений примерно в 100 раз превышают затраты на внесение самих этих изменений. Поэтому мы не можем позволить себе слишком часто проводить подобные процедуры.

Д'Агапеев
Software Engineering, p. 72

Можно довольно легко представить убедительные доказательства полной тщетности попыток исчерпывающего тестирования программы или даже их выборочного тестирования. Но как же быть? Роль тестирования теоретически состоит в том, чтобы подвести основание под индуктивное доказательство безошибочности программы. Вы должны убедиться сами и убедить других, причем как можно более решительно, что программа, если она выполнялась несколько раз со специально подобранными данными, будет работать всегда и с любыми данными. Это можно доказать индуктивным способом. Тестирование таких основных случаев иногда можно было бы автоматизировать. В настоящее время это все еще теоретические рассуждения; отметим, что тесты нужно разрабатывать одновременно с программой, а соответствующая методика испытаний является важной частью документации. Теоретические исследования в этой области свидетельствуют о возможности достижения практических результатов, хотя доказательство правильности функционирования программы — это трудоемкий и дорогостоящий процесс. Однако для определенных ключевых участков программы это нельзя считать излишеством.

Хоар
Software Engineering Techniques, p. 21.

Сложность программ зачастую сильно преувеличивают, и ряд правильно подобранных тестов может полностью решить проблему тестирования. Задача состоит в том, чтобы выбрать эффективные тесты, а не доказывать правильность алгоритма, поскольку это целесообразно делать после соответствующего выбора тестов.

Перлис
Software Engineering Techniques, p. 21.

1) См. примечание на стр. 7

Тестирование выявляет только наличие, но никак не отсутствие ошибок.

Дейкстра
Software Engineering Techniques, p. 21.

Программисты называют свои ошибки неточностями, делая хорошую мину при плохой игре; иначе нельзя было бы психологически оправдать такое громадное число «ляпов»!

Хопкинс
Software Engineering Techniques, p. 23.

Отлаженная программа — это программа, для которой пока еще не найдены такие условия, в которых она окажется неработоспособной,

Огден
Из неопубликованных заметок.

7.0. Введение

Тестирование, отладка и сопровождение машинных программ — это в основном «терра инкогнита». В настоящее время по этому вопросу существует, по-видимому, очень мало литературы, мало или совсем нет исследовательских работ, недостаточно университетских курсов — и очень слабое представление о серьезности проблемы.

Одна из целей данной главы — дать более полное представление об этой проблеме, а также оценку важности усилий по тестированию программ. Кроме того, мы попытаемся рассмотреть и обсудить существующие методы и стратегические принципы, которые могут быть применены для тестирования. Мы укажем также некоторые конкретные вопросы, которые в настоящее время еще не нашли решения.

7.1. Понятия и определения

Прежде чем рассматривать некоторые способы тестирования, попытаемся определить основные термины и понятия, которые здесь будут использованы. Мы не будем приводить «официальные» определения из словарей или другой справочной литературы; более полезно, по нашему мнению, описать эти термины в том смысле, в котором их применяют программисты в своей повседневной работе.

7.1.1. Модуль

Модуль часто определяется так же, как «программа», т. е. как набор команд, достаточных для реализации некоторой завершенной функции. Как мы видели в гл. 3, многие программисты считают термины «модуль» и «подпрограмма» синонимами; мы видели также, что мо-

дуль иногда определяют, исходя из физических ограничений. В некоторых организациях, например, принято считать, что модуль должен содержать не более, скажем, 50 операторов. Во многих случаях, крометого, подразумевается, чтомодуль полезен только тогда, когда он в сочетании с другими модулями используется для образования более крупной единицы (скажем, программы).

7.1.2. Программа

Программу часто пытаются представить как не поддающееся строгому определению произведение искусства, порожденное не без колдовства и черной магии. Более серьезные специалисты в этой области, однако, говорят, что программа является последовательностью команд, которая в сочетании с соответствующими данными и управляющей информацией обеспечивает реализацию некоторого четкого определенного алгоритма. Зачастую под программой понимают такую минимальную последовательность команд, которая можетфункционировать без взаимодействия сдругими программами.

7.1.3. Наборы программ и подсистемы

Подсистема — это набор программ, организованных так, чтобы реализовать многофункциональный алгоритм, превышающий по сложности то, что было бы возможно в обычном случае выполнить с помощью одной программы. Часто встречаются также такие понятия, как *иерархии* подсистем, или уровни подсистем.

7.1.4. Система

Термин «система» нередко применяется для определения некоторого комплекта программ, поставляемых изготовителем вместе с аппаратными средствами ЭВМ; примерами систем могут служить и «операционная система» и «система управления данными». В более широком смысле под системой понимается набор программ и (или) подсистем, обеспечивающий реализацию важных и взаимосвязанных прикладных или каких-либо других функций. Поэтому нам часто приходится слышать такие выражения, как «система составления платежных ведомостей», «система приема и оформления заказов» и т. д.

7.1.5. Надежность программных средств

Шуман [5] определяет надежность программных средств как вероятность того, что данная системная программа в течение заданного периода времени будет безошибочно работать на машине, для которой она создана, при условии, что она используется с учетом ее конструктивных возможностей и ограничений. Надежность часто

выражается такими количественными показателями, как средняя наработка на отказ и средняя наработка до (первого) отказа, или степень сохранности базы данных. Эти показатели, естественно, в гораздо большей степени применимы к системе, чем к одиночной программе или модулю.

7.1.6. Ошибка

В работе [1] Хопкинс просто ставит знак равенства между ошибкой и неточностью. На профессиональном жаргоне, однако, под ошибками обычно подразумеваются огрехи в программе, которые могут проявляться как «постоянные» ошибки (повторяющиеся, типа стойкого отказа) или кратковременные ошибки (типа сбоя).

7.1.7. Ляп

Слово ляп — одно из тех жаргонных выражений, которые получили широкое распространение в словаре специалистов промышленных предприятий и академических институтов Кембриджа и Беркли. Термин «ляп» никогда не имел четкого определения, однако обычно под ним понимают некоторую непредвиденную особенность машинной программы (или в большинстве случаев технического задания и конструкции программы), которая превращает ее в неэффективную, неизящную или неудобную в работе. Отметим, что существует различие между ошибкой и ляпом. Термин «ошибка» означает, что программа работает не так, как того требует техническое задание, а «ляп» — что само техническое задание (или какие-то качества программы, не оговоренные техническим заданием) оказалось непродуманным и некорректным.

7.1.8. Тестирование

Хетцель [10] определяет тестирование как набор процедур и действий, предназначенных для демонстрации правильности работы программы в заданных режимах и внешних условиях. Таким образом, цель тестирования — выявить наличие ошибок или убедительно продемонстрировать отсутствие таких ошибок. Важно различать тестирование и сопутствующее понятие «отладка», которое определяется ниже.

7.1.9. Доказательство правильности программы

В последние несколько лет наблюдается растущий интерес к разработке принципов строгого *доказательства* правильности программы. Это делается обычно безотносительно к внешним условиям в которых работает программа, с помощью набора определений, ак-

сиом и теорем. Можно надеяться, что со временем будут разработаны методы *автоматизации* (полной или частичной) процедуры доказательства правильности программы, но пока в этой области еще много нерешенных проблем. Как мы указывали в гл. 4, трудность доказательства правильности программы сильно зависит от сложности программы и от числа взаимосвязей между ее компонентами; поэтому на структурное программирование, которое призвано уменьшить сложность программ, возлагаются большие надежды и с точки зрения упрощения процесса доказательства работоспособности программ.

7.1.10. Апробирование, верификация и аттестация

Термины «апробирование», «верификация» и «аттестация», хотя они не столь употребительны, как «тестирование» и «отладка», время от времени применяются в обсуждениях проблем контроля работоспособности и качества программ. По определению Хетцеля, *апробирование* — это демонстрация логической правильности программы для *заданных внешних условий* с помощью некоторого набора процедур и действий. *Верификация* — это демонстрация логической правильности программы для данных испытательных условий (которые могут соответствовать или не соответствовать реальным условиям, в которых в конце концов будет работать программа) с помощьюнекоторого иногонабора процедур и действий. Наконец, *аттестация* определяется как подтверждение работоспособности и эффективности программы, основанное на личном опыте, отзывах коллег или полученное с помощью специально организованного набора процедур и действий.

7.1.11. Отладка

Отладка — это набор процедур и действий (ряд которых может быть связан с ЭВМ, а другие — нет), начинающихся с выявления самого факта ошибки и заканчивающихся установлением точного места и характера этой ошибки. По мнению некоторых программистов, отладка включает также период времени, в течение которого ошибка исправляется, а считать отладку законченной можно реально только тогда, когда удается ко всеобщему удовлетворению продемонстрировать, что программа с исправленной ошибкой *работает правильно*.

7.1.12. Сопровождение

Термин «сопровождение» (обслуживание в эксплуатации) применяется обычно для описания всех действий, связанных с исправлением ошибок, доработками с целью обеспечения совместимости с но-

выми операционными системами и (или) трансляторами и проведением *небольших* усовершенствований машинной программы. Различия между небольшими и значительными изменениями, как правило, качественные, но в большинстве случаев такое разграничение не вызывает трудностей. Один вид работы вполне правомерно считать сопровождением, тогда как другой относится уже к категории разработки.

7.2. Масштабы проблемы тестирования

Почти каждый программист (или руководитель отдела программирования) так или иначе видел, слышал или испытал на собственном опыте последствия многочисленных ошибок машинных программ; некоторые из таких историй можно отнести к разряду юмористических, однако наши вычислительные системы выполняют все более важные для общества функции (примером могут служить системы управления воздушным транспортом), поэтому результаты плохой отладки и тестирования программ становятся все более и более серьезными. Несмотря на это, большинство программистов, многие руководители отделов программирования и почти все администраторы совершенно недооценивают затраты средств, времени, энергии и усилий в части планирования, которые необходимы для тестирования программ. Они также нередко недооценивают сложность отладки и, по-видимому, находятся в блаженном неведении относительно важности и сложности квалифицированного сопровождения. Поэтому могут оказаться полезными некоторые статистические данные, иллюстрирующие, насколько серьезными являются эти проблемы.

Например, количество времени, которое необходимо для адекватного тестирования программ (и систем), по оценкам различных специалистов, составляет от 30 до 50% общего времени разработки проекта или даже большую долю. Действительно, было подсчитано, что почти 80% денежных средств, затраченных на проект «Аполлон» Национальным управлением по аэронавтике и исследованию космоса, были выделены на различные виды испытаний. Конечно, небольшую хорошо разработанную программу, вероятно, удастся проверить достаточно полно, затратив на это менее 30% общего времени разработки; тем не менее есть что-то обескураживающее в том, что по-прежнему встречаются планы-графики, где отводится два года на разработку и реализацию сложной системы реального времени и всего лишь один месяц на тестирование этой системы!

Число ошибок, остающихся *невыявленными* в больших программах и системах (после предположительного окончания их отладки и тестирования), очень велико. В работах [2 и 3] сообщается, что каждая новая версия операционной Системы OS/360 содержит свыше тысячи ошибок; Гилб, консультант по вычислительной тех-

нике в Осло, Норвегия, заявляет, что он насчитал свыше *11 тыс.* ошибок в недавней версии OS! Сам автор исследовал несколько операционных систем, поставляемых фирмами-изготовителями ЭВМ, и пришел к выводу, что в течение первых нескольких лет эксплуатации в них выделяется относительно постоянное число системных ошибок (см. [3 и 4]). Более тщательная количественная оценка (например, сделанная Шуманом в работе [5]) позволяет считать, чтр число ошибок в большой сложной системе уменьшается, по-видимому, по экспоненциальному закону; однако, по всей вероятности, при этом не учитываются новые ошибки, постоянно вносимые в операционные системы при их текущем сопровождении и развитии.

Как мы видели в гл. 1, тестирование представляет собой серьезную проблему также с точки зрения программиста. Классическая работа Сакмана [6] показывает, что некоторые программисты на отладку и тестирование своих программ затрачивают в 25 раз больше времени, чем их коллеги. Поскольку эта работа базировалась на довольно малой выборке, ряд специалистов поддался соблазну считать, что реальное положение вещей не столь мрачно; однако личные наблюдения автора, связанного со многими вычислительными центрами, говорят о том, что существует по меньшей мере десятикратная разница в показателях эффективности труда лучших и худших программистов, принадлежащих к средней категории.

Аналогичное исследование, описанное Бемом [2, 7] показало, что программист имеет исключительно малые шансы на успех, если он пытается модифицировать работающую программу. Если изменение по объему не превышает 10 операторов исходной программы, он может правильно внести это изменение с первой попытки с вероятностью приблизительно 50%. Если объем изменения достигает 50 операторов, вероятность успеха падает до 20%.

Если тестирование является трудной проблемой, то нет ничего удивительного в том, что и процесс *сопровождения,*— где обычно имеют место собственные циклы отладки и тестирования,— во многих организациях также сопряжен с трудностями. Как мы упоминали в предыдущих главах, обзор [8] показал, что в средней американской фирме 50% бюджета вычислительного центра затрачивается на сопровождение существующих программ; в некоторых более крупных организациях эта цифра достигает 80%!Анеофициальное обследование, предпринятое одной из ведущих фирм изготовителей вычислительных машин, выявило следующий интересный факт: одна стандартная программа, написанная неким программистом, после этого сопровождалась и дорабатывалась десятью поколениями преемников автора, пока, наконец, не была выброшена и переписана заново!

7.3. Уровни сложности тестирования

Обсуждение тестирования обычно выявляет радикальное расхождение во взглядах различных групп программистов; после дальнейших вопросов часто оказывается, что их взгляды в значительной степени зависят от сложности программ, с которыми они имеют тело. Поэтому, чтобы взглянуть на проблему тестирования с другой стороны, интересно предложить классификацию программ по категориям сложности тестирования. Приведённые ниже категории в достаточной степени произвольны, но они, по-видимому, соответствуют реальным группам приложений, которые можно выделить в вычислительной технике.

7.3.1. Простые программы

Для целей нашего обсуждения *простая* программа определяется как программа, которая

1. Имеет длину исходного текста менее 1000 операторов.
2. Как правило, пишется одним программистом за шесть месяцев или менее;
3. Обычно не имеет взаимосвязей с другими программами или системами.

Несомненно, в вычислительной технике существуют самые разнообразные программы такого типа. Это небольшие прикладные программы научно-технического характера, написанные инженерами для решения относительно простых числовых задач; небольшие сервисные подпрограммы для экономических приложений; небольшие программы для автоматического составления отчетов; подавляющее большинство учебных программ, разработанных на курсах по вычислительной технике, и т. д.

По поводу «простых» программ важно заметить следующее: обычно не имеет значения, каким образом они тестируются,— зачастую даже не имеет значения, подвергаются ли они тестированию вообще! Эти программы, согласно их определению, реализуют относительно несложные, если не совершенно тривиальные, функции и часто представляют собой результат непосредственного кодирования. Благодаря этому программисту будет довольно трудно внести много ошибок в такие программы. К тому же большинство подобных программ не имеет для организации-пользователя критического значения, и поэтому даже их неработоспособность, как правило, не приводит к драматическим последствиям.

Все это позволяет понять, почему чаще всего не проявляется большого интереса к тонким вопросам тестирования со стороны программистов-разработчиков «простых» программ. Они, по-видимому, считают, что могут буквально подавить ошибку в своих программах — практически «грубой силой»,— не прибегая к каким-

либо логически организованным алгоритмам тестирования. Кроме того, они, по-видимому, знают, что, если даже ошибка действительно встретится после окончания тестирования, они будут иметь полную возможность ее исправить.

7.3.2. Программы средней сложности

Следующая категория программ включает программы, каждая из которых

1. Содержит до 10 000 операторов исходного текста.
2. Разрабатывается обычно одним — пятью программистами за менее чем двухлетний срок.
3. Насчитывает мало взаимосвязей с другими системами или вообще их не имеет.
4. Обычно состоит из 10—100 модулей.

Неофициальные обзоры, содержащиеся в нескольких курсах по программированию, которые читал автор, показывают, что к этой категории относится подавляющее большинство машинных программ. Большая часть простых экономических прикладных задач (бухгалтерский учет, прием и оформление заказов, расчеты заработной платы, управление запасами и т. д.) требует для своей реализации, скорее всего, менее 10 тыс. операторов Кобола, абольшинство научно-технических прикладных задач — менее 10 тыс. операторов Фортрана. Даже системные программы (ассемблеры, трансляторы, подсистемы управления данными, программы управления телеобработкой и т. д.) зачастую требуют менее 10 тыс. операторов языка ассемблера (хотя они могут занимать значительно больше 10 тыс. ячеек памяти для размещения буферов, очередей, таблиц и т. д.).

Это не означает, что подобные программы по своей природе тривиальны. Тот факт, что для их разработки требуется один-два года, означает, что работа по их тестированию может занять 6—12 мес. В то же время любые проблемы, которые могут встретиться в процессе испытаний, обычно удается преодолеть одним напором, причем наихудшие последствия в этом случае — сорванный на какое-то время срок ввода в эксплуатацию,— например, программа, разработку которой следовалозакончить за 18 мес, потребовала целых 24 мес. Аналогично, когда законченная программа вводится в эксплуатацию, любые оставшиеся в ней ошибки приводят к нарушениям работоспособности, неприятным, но в принципе не опасным. В большинстве случаев ошибка, обнаруженная в процессе рабочего прогона в понедельник, может быть устранена в понедельник же вечером, так что во вторник программа может снова использоваться для рабочего прогона.

Справедливо, конечно, и утверждение, что некоторые из программ, содержащих 10 тыс. операторов, сложны по своей сути, и

иногда справедливо также то, что ошибки в этих программах вызывают катастрофические последствия; тем не менее можно считать, что большинство таких программ относительно просты. Вследствие этого программисты, занимающиеся разработкой подобных проектов, к сожалению, редко ощущают настоятельную необходимость тщательно изучать и применять улучшенные методы проектирования и программирования (например, нисходящую разработку программ и структурное программирование). Аналогичным образом, они редко ощущают необходимость строить планомерное и всестороннее тестирование своих программ. Можно не сомневаться, что нисходящее проектирование, структурное программирование и некоторые методы тестирования, излагаемые в этой главе ниже, оказались бы весьма полезными. К сожалению, как показывает опыт автора, подобные методы, если в них нет абсолютной необходимости для успеха проекта, будут чаще всего игнорироваться программистами, которые вполне довольны своими старыми и привычными методами.

7.3.3. Сложные программы

Категорию сложных программ можно определить так:

1. Каждая из них содержит до 100 тыс. операторов исходного текста.
2. Обычно разрабатывается группой из 5—20 программистов в течение двух-трех лет.
3. Состоит йз нескольких подсистем.
4. Часто взаимодействует с другими системами.
5. Обычно состоит из 100—1000 модулей.

Отметим, что эта категория включает программы, размер которых лежит в диапазоне 10 тыс.— 100 тыс. операторов; этот диапазон может показаться произвольно большим, однако следует напомнить, что основным критерием является сложность конечной программы, независимо от ее размера. Большинство программистов интуитивно согласятся с тем, что программа примерно такого размера будет, вероятнее всего, сложной; большинство руководителей согласятся с тем, что достаточно большой проект, требующий для своего осуществления более одной группы программистов (и часто более одного уровня координации работ), является, по интуитивной оценке, более сложным проектом, чем программы упомянутых выше категорий. Руководители учитывают также, что длительность разработки такого проекта достаточно велика, поэтому (а) существует большая вероятность, что один или несколько программистов уволятся с работы до завершения проекта, и (б) пользователи будут иметь достаточно времени для того, чтобы продумать и предложить некоторые желательные изменения и усовершенствования про-

граммы, причем будут затем настаивать, чтобы эти изменения были внесены в первоначальную программу, и как раз тогда, когда уже начинается ее тестирование!

Еще важнее рассказать о последствиях ошибок в программах «сложной» категории. Прежде всего после официального ввода в эксплуатацию в программе, по всей вероятности, будет оставаться еще большее число ошибок просто потому, что программа обширнее и сложнее (чем программы предыдущих категорий) и в связи с этим несколько хуже поддается классическим способам тестирования. Далее, когда ошибка проявляет себя (часто в середине рабочего прогона), ее последствия обычно более серьезны, чем для программ предыдущих категорий. Больше машинного времени будет потеряно, интересы большего числа пользователей окажутся затронутыми, более остро встанет вопрос о быстром исправлении ошибки. Усугубляет положение и то, что ошибку зачастую невозможно быстро исправить; ее внешние признаки могут быть известны (например, программа «выбрасывается», не доходя до конца, или выдает неправильные результаты), тогда как характер и место ошибки в программе установить не удается. Ввиду сложности программы локализация и исправление ошибки вполне могут занять два-три дня (а в некоторых случаях — недели и даже месяцы!).

Если такие ошибки обнаруживаются до того, как программа официально передается в эксплуатацию, они, скорей всего, будут исправлены. В конце концов именно для этого выполняются системные тесты, параллельные прогоны и контрольные задачи для приемо-сдаточных испытаний, которые обычно придаются подобным программам. Однако принципиальная сложность программы и «неуловимый» характер ошибок (а также их кажущаяся неисчерпаемость и разнообразие) нередко затягивают процесс тестирования далеко за пределы конечного срока, первоначально установленного для проекта.

Последствия подобных проблем могут быть довольно серьезными. Разработка программы часто завершается с значительной задержкой относительно планового срока; это раздражает заказчиков и приводит обычно к тому, что упускаются экономические выгоды, которые предположительнодолжна была быдать программа. Невыявленные ошибки (которые, как правило, являются существенной особенностью таких программ) приводят в дальнейшем к большим потерям машинного времени, срывают производственные планы и вызывают сильное неудовольствие пользователей — и, быть может, серьезные экономические неприятности для организации — разработчика программы. Следует также указать, что именно при разработке программ этой категории часто приходится видеть, как прекращаются работы по незаконченному проекту; прекращаются просто потому, что организация-разработчик оказалась не в состоянии создать приемлемо надежную программу.

Сказанное позволяет сделать следующий вывод: мы можем иногда позволить себе игнорировать новые эффективные способы тестирования (и эффективные способы проектирования программ), когда речь идет о простых программах и программах средней сложности; в случае сложных программ выбор неоптимальных способов тестирования может оказаться причиной полной катастрофы проекта. Сейчас к такому уровню сложности приближается все большее число экономических и научно-технических прикладных программ (а также ряд системных программ, например, больших трансляторов), поэтому средства тщательного тестирования становятся необходимостью, в то время как для программ ранее охарактеризованных категорий они были только желательны.

7.3.4. Сверхсложные программы

К счастью, только относительно малое число существующих программ относится к категории «сверхсложных». По определению это программы, которые

1. Содержат до 10^6 операторов исходного текста.
2. Разрабатываются коллективом из 100—1000 программистов в течение нескольких лет.
3. Для своей последующей доработки, развития и сопровождения требуют привлечения специалистов, которые не входят в первоначальный коллектив разработчиков.
4. Как правило, состоят из 1000—10 000 модулей.
5. Как правило, состоят из нескольких основных подсистем со сложными взаимосвязями между ними; возможны также сложные взаимосвязи с другими отдельно разработанными системами.
6. Часто предусматривают дополнительные сложные режимы работы, такие, как вычисления в реальном масштабе времени, телеобработку данных и многопрограммную работу.

До последних нескольких лет столь крупные проекты были прерогативой ведущих фирм — изготовителей ЭВМ (примером может служить разработка OS/360 фирмой IBM) и некоторых честолюбивых правительственных и военных управлений. Недавно, однако, небольшое, но постоянно растущее число промышленных организаций начали работать над проектами такого размера; напомним, что, согласно определению, программы этой категории содержат от 100 тыс. до миллиона кодированных строк. Перспективы достичь этого уровня размеров и сложности появились теперь у крупных банковых прикладных систем, крупных прикладных систем страховых фирм, у «интегрированных» информационно-управляющих комплексов и некоторых других систем. Именно *благодаря* появлению таких сверхсложных программ и систем мы наблюдаем теперь здравый подход к тестированию (т. е. подход, характеризующийся четким пониманием проблемы, осторожностью и предусмотритель-

ностью). Многим организациям, приступающим к таким проектам, стало совершенно ясно, что существуют *весьма малые шансы* на успех, если твердо не придерживаться как жестких принципов конструирования программ (например, модульного и структурного программирования), так и жестких принципов тестирования. В то же время явная важность и необходимость тестирования обусловила разработку некоторых автоматизированных способов и средств, описанных в разд. 7.8.

7.3.5. Гиперсложные программы

Наконец, упомянем категорию программ, которые можно классифицировать только как «гиперсложные». Такие программы:

1. Содержат от миллиона до десяти миллионов команд.
2. Разрабатываются обычно коллективом, превышающим тысячу программистов, причем длительность разработки составляет несколько лет, а часто приближается к десятилетию или более.
3. Почти всегда включают вычисления в реальном масштабе времени, телеобработку данных и прочие сложные режимы.
4. Зачастую связаны с критическими процессами, например с управлением движением воздушного транспорта или с противовоздушной обороной.
5. Обычно характеризуются исключительно высокими требованиями по надежности (например, в техническом задании на один подобный проект в Австралии для системы указана наработка на отказ, равная 47 годам).

Излишне говорить что очень немногие организации когда-либо брались за подобные проекты. В настоящее время их число ограничивается, видимо, правительственными и военными управлениями. Лично автор считает, что наши сегодняшние знания принципов конструирования и тестирования программ недостаточны для того, чтобы успешно разработать систему такого масштаба; тем самым создается сложнейшая дилемма, поскольку упомянутые выше правительственные и военные управления, по их словам, *должны* реализовать подобные системы, если они желают обеспечить удовлетворение постоянно растущих потребностей общества. В некоторых случаях успех или неуспех системы — момент, скорей всего, спорный, например, если речь идет о системе противовоздушной обороны, которая, хотелось бы надеяться, никогда не будет использована по назначению на практике; в то жа время некоторые системы, планируемые в настоящее время гражданскими правительственными организациями, будут предположительно использоваться, так что последствияплохоготестированиямогутбыть довольно серьезными.

Есть какая-то ирония в судьбе разработок подобных систем. Многие из них связаны с секретной деятельностью правительственных или военных организаций, поэтому коллективы разработчиков

зачастую изолируются от своих коллег, работающих в области создания экономических или научно-технических систем, и тем самым теряют возможность воспользоваться опытом, приобретенным специалистами по обработке экономических данных. В то же время богатый опыт специалистов по вычислительной технике правительственных и военных организаций (уже свыше двадцати лет разрабатывающих сложные системы, ряд которых оказался исключительно удачным) погребен в секретных архивах, очевидно, для того, чтобы не предоставлять все материалы гражданским специалистам, которые могли бы их использовать в своей работе. Таким образом, можно считать почти неизбежным, что гражданские специалисты по вычислительной технике будут сталкиваться с теми же самыми ошибками и неприятностями, которые возникали у военных, когда через пять или десять лет и *они* начнут разрабатывать гиперсложные программы (или системы).

7.4. Виды ошибок, которые должны выявляться при тестировании

Быть может, говорить об этом несколько наивно, но, наверное, нет смысла проводить тестирование, если мы не знаем, *что* мы хотим получить в результате. Как указывает Дейкстра, тестирование выявляет наличие, но не отсутствие ошибок. Но какого рода ошибок? Для большинства программистов этот вопрос кажется тривиальным; что мы еще можем искать, как не *логические* ошибки? Если вычислительные машины и системы рассматривать просто как продолжение наших врожденных (человеческих) мыслительных средств, то, быть может, все программные ошибки фактически являются просто ошибками нашей собственной логики; однако обычно бывает более целесообразно определить несколько *категорий* ошибок и соответственно им строить нашу стратегию тестирования. Поэтому ниже мы перечисляем виды ошибок, которые обычно встречаются в машинной программе.

7.4.1. Логические ошибки

Действительно, логические ошибки *представляют,* как правило, наиболее распространенный вид ошибок; и основная часть наших программных усилий по тестированию вполне оправданно направлена на борьбу с такими ошибками. Для целей нашего обсуждения мы можем считать, что логическая ошибка — это постоянная, повторяющаяся ошибка. Если данный тест (входная тестовая комбинация) выявляет присутствие некоторой ошибки, то этот же тест, проведенный для программы второй раз, должен выявить ту же самую ошибку аналогичным образом.

7.4.2. Ошибки документации

Существуют некоторые аспекты прикладного программирования, где ошибка документации может привести к столь же серьезным последствиям, как и логическая ошибка. В большинстве случаев нас должны сильнее беспокоить ошибки в документации для пользователя — документации, в которой пользователю даются инструкции, как готовить исходные данные для программы, как управлять работой программы и как применять и интерпретировать выходные данные программы. Существуют, однако, и другие ситуации, где критическими можно было бы считать ошибки в *технической* документации.

7.4.3. Ошибки перегрузки

Часто важно испытать программу таким образом, чтобы обнаружить, что произойдет, если различные внутренние таблицы, буферы, элементы очередей или другие области памяти будут заполнены целиком или даже переполнены. Эта проблема носит особенно критический характер при тестировании многих систем коллективного пользования и реального времени (например, что случится, если все абоненты системы будут одновременно печатать на своих терминалах входные сообщения?), но может быть столь же важной и для многих программ, ориентированных на пакетную обработку.

7.4.4. Временные ошибки

Ошибки этой категории обычно относятся только к системам реального времени. В данном случае мы имеем в виду логические ошибки, которые *нельзя* легко *повторить;* эти ошибки, как правило, зависят от временных режимов или от совпадающих сочетаний событий внутри программы. Для программы, работающей не в реальном времени, мы можем, как правило, утешаться тем, что существует некоторое конечное число случаев, подлежащих проверке (хотя зто число столь велико, что мы обычносовершенно не в состоянии провести исчерпывающее тестирование). Для системы реального времени, однако, число возможных временных ситуаций является (по крайней мере на первый взгляд) бесконечным.

7.4.5. Ошибки быстродействия и емкости

Ошибки этой категории также имеют существенное значение лишь для систем реального времени, хотя, по-видимому, в этом плане следует проверять гораздо большее число программ пакетной об-

работки. Здесь мы имеем в виду *быстродействие* программы; программа может выдавать правильный результат, но затрачивать неприемлемо большое количество машинного времени ЦП или требовать для своего выполнения чрезмерно большого объема оперативной памяти, дисковой памяти и т. д. Этот фактор является критическим для многих систем коллективного пользования, поскольку быстродействие программы абонент зачастую сразу же видит по времени реакции на его запросы. Для программы пакетной обработки у нас может также возникнуть необходимость задать (а затем проверить), что программа должна позволять обрабатывать одно событие (одно изменение) в секунду, что она должна занимать не более 100 тыс. байт памяти и т. д.

7.4.6. Ошибки при возврате к контрольной точке и восстановлении

Для ряда программ довольно большое значение имеетвозможность возврата к некоторой точке и восстановления режима счета в случае аппаратного отказа (или, быть может, отказа программного обеспечения). Программа, где такие возможности не предусмотрены, может вызвать потерю нескольких часов машинного времени или в случае системы коллективного пользования замешательство среди абонентов и полный хаос. Тестирование в этом плане должно гарантировать, что выполнение программы можно возобновить с некоторой контрольной точки, файлы не портятся, полная процедура восстановления может быть осуществлена за разумное время и что при этом не будут затронуты интересы пользователей, операторов ЭВМ и других лиц.

7.4.7. Аппаратные ошибки и ошибки системных программных средств

В большинстве случаев программист считает, что в его обязанности не входит подтверждение правильности работы аппаратуры и операционной системы фирмы-поставщика, и чаще всего ему и не приходится этого делать. Если, однако, проводятся испытания системы в целом и если эта система предназначается для поставки пользователю, не связанному с вычислительной техникой, то кто-то должен взять на себя ответственность и гарантировать, что аппаратура и операционная система действительно *работают,* поскольку пользователь вряд ли с сочувствием примет, например, такое объяснение программиста: «Но это не моя вина, что система оперативного оформления заказов отказала, в результате чего потеряны заказы за целый день,— это была аппаратная ошибка».

7.4.8. Ошибки несоблюдения стандартов

Наконец, некоторые специалисты считают, что программы следует проверять на соответствие требованиям различных стандартов программирования. Это позволит удостовериться в том, что программы построены по модульному принципу, имеют хорошие комментарии, не содержат нестандартных операторов и т. д. Этому вопросу уделяется все большее и большее внимание со стороны организаций, которые начинают понимать, насколько это важно для успешного сопровождения в эксплуатации.

7.5. Этапы тестирования

7.5.1. Необходимость поэтапного тестирования

Все программы, кроме самых небольших, слишком сложны, чтобы их можно было проверить за один прием; если мы хотим иметь достаточные шансы на успех, мы должны проводить испытание программы поэтапно. Велика также роль разумного планирования — в противном случае программист зачастую работает в режиме «штурмовщины», а в спешке допускается много промахов. Здесь уместно еще раз напомнить знаменитое высказывание Дейкстры по поводу тестирования («тестирование выявляет наличие, но не отсутствие ошибок»). Если мы *не обнаружили* каких-либо ошибок, проверяя крупную и сложную программу, то с большой вероятностью можно считать, что испытания проводились слишком тривиально.

«Тривиальное тестирование» — реальная опасность для многих проектов систем математического обеспечения. Нередко тривиальное тестирование может приводить к тому, что программа или вычислительная система передается заказчику до того, как она реально готова к эксплуатации. Учитывая нашу склонность к недооценке требуемых для разработки систем матобеспечения времени и ресурсов (этатема выходит за рамки нашего обсуждения), можно не удивляться тому, что руководители проекта и программисты жертвуют полнотой испытаний, стараясь любой ценой обспечить выполнение работы в запланированный срок.

Кроме того, следует указать, что тщательное планирование этапов тестирования важно для надлежащей подготовки руководителей, пользователей, обслуживающего персонала ЭВМ, а также других участников работы к жестким режимам периода испытаний.

Интересно отметить, что существуют в основном два различных подхода к реализации идеи «поэтапного тестирования». В качестве классической схемы тестирования часто принимается восходящее тестирование в противоположность относительно новому принципу нисходящего тестирования. Классическое восходящее тестирование рассматривается в подразд. 7.5.2, а нисходящее — в подразд. 7.5.3.

7.5.2. Восходящее тестирование

Классической схеме тестирования недавно было дано имя восходящего тестирования главным образом для того, чтобы отличить этот способ от нисходящего тестирования, о котором первоначально упоминалось в гл. 2. В любых случаях, за исключением самых малых программ, способ восходящих испытаний обычно требует трех этапов: тестирования модулей, тестирования подсистем и тестирования системы. В экономических приложениях часто бывает и четвертый этап — этап приемо-сдаточных испытаний.

Тестирование модулей. Тестирование модулей обозначается также терминами «автономная проверка», «проверка программы», «одноходовая проверка» и рядом других. Независимо от названия, однако, здесь речь идет о начальном уровне тестирования, обычно требующем работы одного программиста в течение периода времени не более месяца (по данным некоторых вычислительных центров, иногда этот период может составлять всего один-два дня).

Тестирование модулей подразумевает автономную проверку, т. е. имеется в виду, что модуль может быть проверен при отсутствии других модулей системы. Однако, чтобы сделать такое тестирование практичным и простым, нам может потребоваться база тестовых данных и прочие элементы окружения, в котором в конце концов будет работать программа. Частично благодаря этому предоставляются такие широкие возможности применения автоматизированных способов тестирования, рассматриваемых в разд. 7.7.

Для успешного тестирования модулей (и в первую очередь, чтобы облегчить последующие этапы тестирования) мы должны стремиться к тому, чтобы оно было исчерпывающим. Кроме того, мы должны быть в состоянии *продемонстрировать,* что тестирование проведено исчерпывающим образом (одна из жизненных сцен, исполненных подлинного трагизма,— это программист, говорящий: «Ну, наконец-то, я, *кажется,* проверил все возможные случаи...»). Отметим, что это еще один довод в пользу применения автоматизированных средств тестирования, описанных в разд. 7.7. Отметим также, что для исчерпывающего тестирования модулей необходимо, чтобы разработанная программа состояла из небольших, функционально независимых модулей.

При тестировании по восходящему принципу полная и тщательная проверка модулей является ключом к успеху. Вследствие этого специалистам по вычислительной технике настоятельно требуются более надежные статистические данные, помогающие организовать более эффективное тестирование. Например, было бы исключительно полезно получить статистические данные, говорящие о том, какое среднее число попыток тестирования (выходов на машину) требуется для среднего модуля (приведенного, например, к числу попыток **тестирования на тысячу операторов). Аналогично необходимы более**

надежные статистические данные, говорящие о том, сколько, вероятнее всего, проходит реального времени, прежде чем модуль оказывается полностью проверенным (напомним здесь о расхождениях во времени отладки и тестирования, о которых пишет Сакман в своем классическом исследовании методов работы программистов).

Тестирование подсистем. Цель тестирования на этом следующем уровне — проверить сопряжение между различными модулями в программе. Здесь мы ожидаем, что обнаружится ряд *логических* ошибок и *ошибок сопряжения,* в противоположность относительно тривиальным ошибкам кодирования, которые обычно выявляются на уровне тестирования модулей. Для большой и сложной программы или системы предусматривают, как правило, много подуровней тестирования подсистем, причем производят тестирование по иерархии подсистем в направлении «снизу вверх».

Было бы желательно осуществлять исчерпывающую проверку и на этом уровне, однако это, вероятнее всего, окажется нереальным. Здесь опять-таки нам хотелось бы иметь возможность продемонстрировать полноту нашего тестирования. Программист должен иметь возможность объявить во всеуслышание, насколько исчерпывающую проверку в действительности обеспечивает тестирование его подсистем, а если эта проверка не исчерпывающая, то должен точно указать, какие части его программы проверены. Тот факт, что программист нередко даже не знает, какие части его программы действительно проверяются при тестировании, является еще одним доводом в пользу автоматизированных способов тестирования, предлагаемых в разд. 7.7.

Тестирование системы. Очевидно, что на этом уровне мы проверяем систему в целом; мы ожидаем, что здесь будут выявляться наиболее тонкие ошибки, например ошибки сопряжения, сложные ошибки управления и логики, ошибки восстановления, ошибки быстродействия и емкости, а также временные ошибки.

Исключительно важно установить хорошие критерии успешного тестирования на уровне системы, т. е. необходимо заранее условиться о том, в какой момент можно прекратить тестирование системы. В качестве критериев можно было бы выбрать, например, следующие:

1. Число событий или тестовых примеров (ситуаций), которые успешно проверены.
2. Число часов правильного функционирования.
3. Процент модулей или операторов исходного текста, проверенных при тестировании.

Как мы отмечали выше, нужны более достоверные статистические данные, чтобы вычислительные центры могли правильно определять плановые сроки для различных этапов тестирования; длительность тестирования систем, по-видимому, постоянно недооценивается. Считается обычно, что тестирование системы займет 25—30% обще-

го времени разработки среднего проекта; и все же встречаются еще организации, которые оптимистически планируют провести испытания системы за два месяца для проекта, разрабатывающегося в течение трех лет.

Приемо-сдаточные испытания. Последний этап, обычно встречающийся при восходящем тестировании,— это приемо-сдаточные испытания; в некоторых организациях этот этап называют «эксплуатационными испытаниями», «испытаниями по программе пользователя», «параллельным тестированием» или другими именами. Цель этих испытаний очевидна: проверив новую систему в целом на функционирование в искусственных условиях (часто на специально подготовленных тестовых данных и контрольных задачах), необходимо затем проверить ее в реальных условиях и сравнить выданные ею результаты с результатами, полученными на существующей вычислительной системе или вручную.

Приемо-сдаточные испытания, безусловно, требуют тщательно спланированной координации усилий и взаимодействия различных групп специалистов внутри организации, где должна работать новая вычислительная система, главным образом пользователей, руководителей (руководителей подразделений обработки данных, руководителей отделов-пользователей, общей администрации) и персонала, обслуживающего ЭВМ в эксплуатации. Не один проект в области вычислительной техники потерпел неудачу единственно лишь потому, что процесс приемо-сдаточных испытаний столь сильно нарушал порядок работы отделов-пользователей, что они в конце концов решали, что им будет спокойнее без новой системы.

При составлении планов-графиков приемо-сдаточных испытаний рекомендуется предусматривать достаточное количество резервного времени, чтобы можно было исправлять ошибки по мере их обнаружения. Другими словами, не следует планировать на весь период приемо-сдаточных испытаний непрерывный круглосуточный прогон, в конце концов программистам обязательно потребуется машинное время второй и третьей смены для исправления ошибок, выявленных в течение первой смены.

7.5.3. Нисходящее тестирование

Как уже указывалось в гл. 2, нисходящее тестирование означает, что мы выделяем основную программу и, допустим, один или два уровня подпрограмм в качестве ядра системы и проверяем отдельно это *ядро;* очевидно, что при этом все или большинство модулей более низкого уровня реализуются вначале как фик тивные подпрограммы. Установив работоспособность ядра системы, мы начинаем наращивать саму систему, доб авляя по одному новому модулю более низкого уровня и проверяя его отдельно. Блок-схема, иллюстрирующая эту процедуру, показана на рис. 7.1.

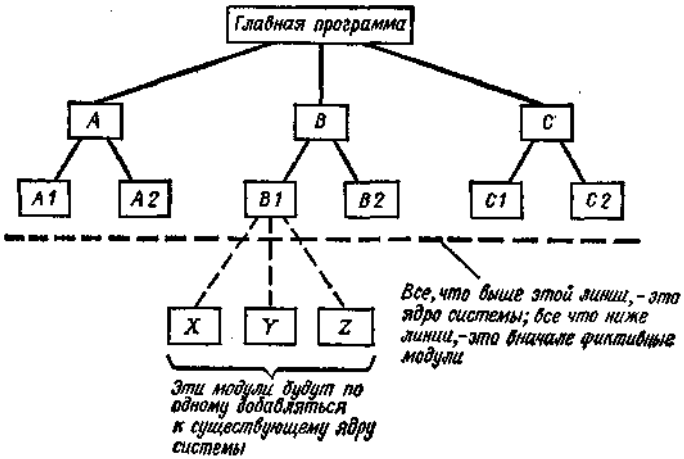


Рис. 7.1. Нисходящее тестирование.

Нужно подчеркнуть, что нисходящее тестирование не следует рассматривать как абсолютно жесткий способ проверки работоспособности программ. Это скорее *методология,* так что конкретный путь тестирования может (и должен) меняться в зависимости от ситуации. В наиболее простых ситуациях при программировании можно без особых затруднении следовать блок-схеме, приведенной на рис. 7.1; однако есть ряд сложных ситуаций, где нужно несколько изменять строгий порядок нисходящего тестирования.

Например, иногда бывает, что нельзя применить идею включения фиктивных модулей вместо модулей, находящихся ниже текущего уровня тестирования. В обычном случае фиктивные модули имеют просто вход и сразу же выход и не выполняют никаких реальных вычислений; но может быть, что они выдают некоторые *постоянные* выходные данные или *случайные* выходные данные; в других случаях может оказаться полезным, чтобы фиктивный модуль формировал сообщение в составе выходного файла,— это позволит программисту установить по распечатке, что к данному модулю было обращение. Однако существуют ситуации, когда фиктивные модули низкого уровня выполняют критические функции, необходимые для тестирования других модулей более высокого уровня; примерами могут служить модули ЧТЕНИЕ и ЗАПИСЬ, встречающиеся во многих программах.

Для ситуаций такого рода существует несколько возможных решений:

1. Несколько критических модулей низкого уровня можно заблаговременно сконструировать, закодировать и проверить автономно, чтобы упростить комплексное тестирование остальных высокоуровневых частей программы.

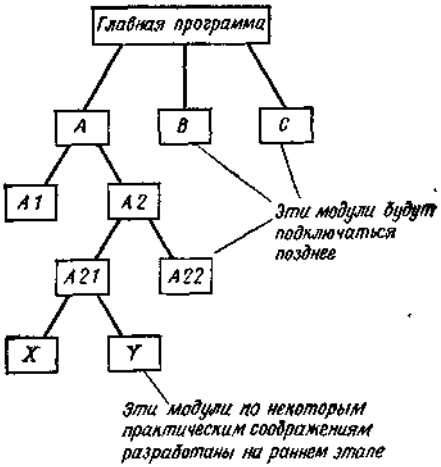


Рис. 7.2. Разновидности нисходящего тестирования.

2. Можно применить некоторую комбинацию способов выходящего и нисходящего тестирования.

3. Можно создать вначале предварительную, или упрощенную, версию модуля низкого уровня. Если модуль ввода-вывода является одним из важнейших, может оказаться целесообразным, например, разработать упрощенный модуль, выполняющий все основные функции по вводу-выводу, но без буферизации, блокирования записей, контроля ошибок, оптимизации и прочих дополнительных средств и усовершенствований. С учетом того, что эти средства и усовершенствования могут быть реализованы в самих модулях низкого уровня, такое решение не противоречит основам методологии нисходящего тестирования.

4. Определенные части программы иногда требуется полностью проверить раньше, чем другие части, как это показывает рис. 7.2. Действительно, ряд возвратных шагов вверх и вниз по «древовидной структуре» программы может оказаться наилучшим способом тестирования.

Некоторые программисты высказывают мнение, что именно модуль низкого уровня может в ряде случаев определять общий успех программы в целом. Если это действительно так, то это может послужить доводом в пользу конструирования, кодирования и тестирования такого модуля низкого уровня в начале работ по проекту (или, быть может, параллельно с разработкой остальных частей программы). После проверки работоспособности самого важного модуля низкого уровня, однако, остальные этапы процесса тестирования могут выполняться по схеме «сверху вниз».

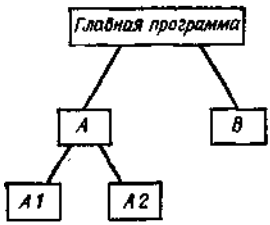


Рис. 7.3. Проблемы нисходящего тестирования. Теоретически модуль A1 следовало бы тестировать путем передачи данных «сверху», т. е. от главной программы, через модуль А. Однако может оказаться сложным сформировать оптимальные тестовые данные, которые, будучи преобразованы в модуле А, обеспечат достаточно полную проверку модуля A1.

Другие программисты указывают, что можно было бы сэкономить значительное количество машинного времени, если подвергать модуль автономному тестированию перед включением его в иерархическую структуру. Это обеспечило бы выявление наиболее тривиальных ошибок кодирования до того, как модуль подключается к ядру программы. Это может быть действительно целесообразным; в то же время следует напомнить, что существующее ядро программы может представлять собой отличную среду или основу для выполнения тестирования нового модуля (примерно в том же смысле, как тест-диспетчер, описанный в разд. 7.7.3).

При этом возникает другая потенциальная проблема. В некоторых случаях может оказаться трудным подготавливать оптимальные тестовые данные при нисходящем тестировании; это иллюстрирует схема на рис. 7.3. Модуль низкого уровня может быть разработан с расчетом на восприятие широкого диапазона входных кодов; по различным причинам задача формирования входных кодов высокого уровня так, чтобы на тестируемый модуль поступали соответствующие входные коды низкого уровня, может оказаться нелегкой. В случае такой структуры более целесообразно, по-видимому, осуществлять тщательное автономное тестирование, обеспечивающее проверку модуля на полном наборе входных кодов. Затем модуль можно включить в иерархическую структуру и обеспечить проверку *интерфейсов* между модулем и остальными частями программы.

Наконец, следует указать, что нисходящее тестирование может оказаться нерациональным для небольших программ, например программ, содержащих менее 500 исходных операторов. Фактически при таком тестировании может быть больше хлопот, чем пользы. То же самое соображение справедливо и относительно нисходящего тестирования некоторых систем. Как показывает рис. 7.4, иногда целесообразно полностью проверить вначале малые подсистемы крупной сложной системы, а затем переходить на нисходящее тестирование, более сложных подсистем.

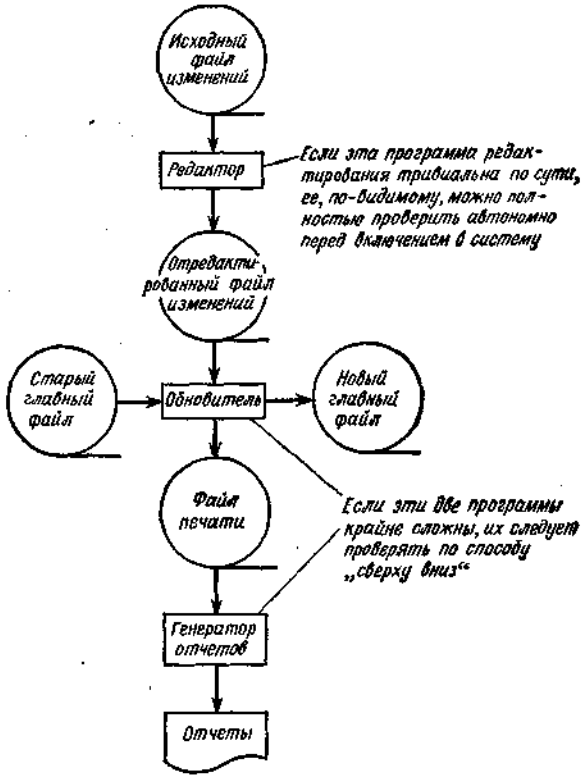


Рис. 7.4. Сочетание способов нисходящего и восходящего тестирования.

Несмотря на эти оговорки, основная методология нисходящего тестирования остается весьма эффективной. Достоинства такого подхода можно суммировать следующим образом:

1. При нисходящем тестировании мы добавляем всего по одному новому модулю при каждом новом цикле проверки; если что-то идет неправильно, то источник и характер ошибки обычно бывают довольно очевидными. При восходящем тестировании, напротив, при каждом новом уровне тестирования в работу вступает все увеличивающееся число модулей, что значительно затрудняет процесс *отладки* (в противоположность процессу тестирования).

2. При нисходящем тестировании большинство основных ошибок интерфейса, управления и логики обнаруживаются в начале процесса тестирования, а частные ошибки кодирования выявляются в конце, а не наоборот. Почти во всех случаях для нас было бы предпочтительно иметь известную уверенность, что основной алгоритм программы работает правильно и что остающиеся дефекты — это просто тривиальные ошибки кодирования.

3. Когда нисходящее тестирование применяется в сочетании с нисходящим проектированием и нисходящим программированием, то можно избежать многих недоразумений в связи с нечеткостью технического задания. Нередко оказывается, что причиной ряда программных ошибок является следующее: два разных программиста по-разному понимают техническое задание на интерфейс.

4. На относительно ранней стадии уже существует *работающая* программа; несмотря на тот факт, что многие из конкретных функций программы могут быть еще не проверены (или даже не реализованы), программу все же можно использовать в работе, демонстрировать пользователям и проводить с ней эксперименты. Трудно переоценить психологические и конъюнктурные преимущества такого подхода при работе с крупной программой, которая в противном случае потребовала бы нескольких месяцев (или лет) тестирования без ощутимых результатов.

Следует отметить, что многие программисты изредка прибегают к нисходящему тестированию, практически не задумываясь о том, что они делают. Во многих случаях этот способ тестирования выбирается лишь потому, что другого выхода нет: план срывается, а нисходящее тестирование кажется единственным путем получения работающей (хотя и неполной) программы. В этом разделе, однако, мы излагали подход к нисходящему тестированию, которое организуется сознательно.

7.6. Конструирование программ с целью облегчения тестирования

Существенные перемены по отношению к тестированию вычислительных систем, кажется, наблюдаются среди многих теоретиков вычислительной науки. Многие теперь вполне уверенно считают, что наилучший (и, быть может, *единственный)* способ сделать тестирование полным и в то же время экономичным — это разрабатывать программу так, чтобы в ней не содержалось ошибок. В действительности здесь имеют место два вопроса. Первый состоит в том, что к моменту времени, когда мы реально готовы выполнять машинную программу, в ней практически не должно оставаться ошибок, т. е. большинство ошибок должно быть устранено путем предварительной ручной проверки работоспособности программы «на бумаге» и визуальной проверки закодированной рабочей программы. Более важно то, что выбранный способ конструирования и кодирования программы должен с самого начала предотвращать проникновение ошибок в рабочую программу. Второй вопрос состоит в том, что программы *всегда будут* содержать ошибки, хотим мы того или нет, но наши программы должны быть разработаны таким образом, чтобы характер и причины ошибок были бы сразу же ясны и ошибки можно было легко устранять.

Многие из таких принципов программирования уже рассматривались в предыдущих главах книги; тем не менее полезно еще раз суммировать их здесь в связи с нашим обсуждением проблем тестирования.

7.6.1. Неоднозначность языков программирования

Некоторые теоретики вычислительной науки указывают, что многие наиболее тонкие и особенно пагубные ошибки в программах можно было бы исключить путем разработки новых языков программирования (или переработки существующих языков), синтаксические правила которых были бы определены более четко. Благодаря этому у программиста оставалось бы меньше возможностей написать что-то такое, чего он на самом деле не имел в виду. В качестве одного из примеров приведем ряд возможных ошибок по недосмотру программиста, работающего на языке ПЛ/1. Если программист не объявит четко спецификации и характер использования своих переменных, то существует нетривиальная возможность, что компилятор сформирует ошибочное объявление, которое со временем доставит много хлопот программисту. В несколько меньшей степени эта же проблема существует даже для Фортрана: при отсутствии другой спецификации переменная, начинающаяся с символа I, J, K, L, M или N, будет рассматриваться как целая величина.

Многие программисты предлагают решение совсем другого рода; они считают, что некоторые распространенные ошибки можно было бы исключить, если бы языки программирования содержали в качестве примитивов реализацию наиболее употребительных процедур. Чтобы прибавить 1 к целой величине I, например, во многих языках программирования обычно применяется инструкция вида

```
I = I + 1
```

Из-за ошибки перфорирования при подготовке данных эта инструкция может довольно легко оказаться преобразованной в инструкцию

```
I = I + I
```

или

```
1 = 1 + 1
```

причем обе инструкции являются синтаксически правильными. Эту ситуацию делает опасной та легкость, с которой программист может пропустить подобную ошибку перфорирования при просмотре листинга своей программы. Ситуацию можно было бы поправить, если бы язык программирования включал, например, инструкцию

```
BUMP I
```

Хотя операторы такого характера интересны и в какой-то степени возбуждают воображение, они, к сожалению, почти не затрагивают среднего программиста. Разработка новых языков программирования или модификация существующих языков — это задача, решение которой, по-видимому, лучше оставить различным комитетам по стандартизации.

В то же время некоторые вычислительные центры решают ту. же самую проблему другим способом: обнаружив, что некоторые языковые средства неоднозначны или часто приводят к ошибкам, организации просто разрабатывают собственные стандарты, запрещающие своим программистам применять эти средства. Например, во многих организациях, работающих на Коболе, запрещают использовать сложные булевы выражения, так что им не приходится иметь дело с операторами типа

```
IF A AND B EQUALS C OR NOT D MOVE X TO Y.
```

смысл которых неоднозначен для читающего их человека, если даже они правильно воспринимаются компилятором с Кобола.

7.6.2. Защита от ошибок как стиль программирования

Об этом шла речь в гл. 6. Основная мысль гл. 6 может быть сформулирована так: вы должны считать при разработке своего модуля, что все, что может быть использовано неправильно, *будет* делаться неправильно. Перефразируя один из законов Мерфи, можно сказать: если, по всей видимости, *ничего* нельзя сделать неправильно, что-то все же обязательно будет не так. При написании программы вам следует руководствоваться следующим правилом: подозревайте всех и каждого. Если ошибки все же *возникнут* при таком подходе, их будет легче локализовать и исправить.

Конкретно мы рекомендуем, чтобы программа контролировала все входные данные, поступающие извне (с перфокарт, ленты, терминалов и т. д.), все обращения из других модулей (это особенно важно), чтение записей из базы данных или файла, сигналы оператора ЭВМ и т. д. Безусловно, можно было бы затратить бесконечное количество времени на контроль всех этих возможных ошибок, но не в этом цель. Мы предлагаем осуществлять в *разумных* масштабах контроль этих потенциальных источников сбоев и отказов.

В гл. 6 предлагаются также некоторые практические приемы программирования, помогающие уменьшить число остающихся в машинной программе ошибок. Рекомендуется, например, при условном переходе проверять все возможные случаи, т. е. ко всем условиям для принятия решения следует добавлять «ни одно из вышеуказанных» событий. Аналогично отмечалось, что программы с большим числом флажков зачастую более гюдвержены ошибкам; это особенно справедливо, если флажки устанавливаются, сбрасы-

ваются и анализируются случайным образом по всей программе. Здесь наше особое беспокойство вызывает то, что значительное число сочетаний или перестановок флажков может быть бессмысленным, а если одно из таких сочетаний встретится (из-за ошибки в какой-то точке программы), поведение программы может оказаться непредсказуемым.

7.6.3. Структурное программирование

Структурное программирование было предметом длительного обсуждения в гл. 4, и здесь нет необходимости повторять его основные принципы. Важно еще раз подчеркнуть, однако, что одна из главных целей структурного программирования — резкое уменьшение числа ошибок проектирования и ошибок кодирования благодаря тому, что программист *вынужден* разрабатывать свою программу как ряд «черных ящиков».

7.6.4. Коллективное программирование

Этот принцип решительно рекомендует Вейнберг в книге [11], называя его «неэгоистическим» программированием. Сама идея не очень нова (одно время это была довольно стандартная форма программирования), ноонавсеже, по-видимому, неслишкомраспространена в большинстве крупных вычислительных центров. Основная идея состоит в том, чтобы поощрять каждого программиста показывать свою программу в исходном виде и на машинном языке *всем* сотрудникам группы по разработке проекта для просмотра и критики. Отметим различие между этим принципом и более стандартным подходом, при котором программа в исходном виде и на машинном языке показывается только *одному* человеку — обычно руководителю проекта или системному аналитику. При коллективном программировании предполагается, что все члены группы разработчиковбудут в курсе дела и будут принимать участие в конструировании и кодировании модулей каждого из коллег. В практике некоторых организаций эта идея развивается еще дальше — здесь ответственность за успех (или неуспех) системы в комплексе возлагается на всю *группу;* следовательно, и ответственность за успех каждого модуля несет группа в целом, а не просто индивидуальный программист, который конструировал и (или) кодировал этот модуль.

Первоначальной причиной применения коллективного программирования,— предложенного Вейнбергом и реализованногофирмой IBM в классическом проекте *Нью-Йорк Таймс* [9],— было стремление устранить ошибки и подвергнуть всю рабочую программу тщательному контролю еще до того, как она попадет в машину. Считалось, что при этом удастся практически избежать системноготестирования и что число остающихся ошибок будет много меньше, чем

при традиционном подходе. Всего этого *действительно* удалось добиться; крометого, такой подход, по-видимому, существенноувеличил производительность труда каждого программиста.

Подобный подход определенно имеет большие перспективы в качестве дополнения к прочим способам тестирования, однако, очевидно, существуют некоторые потенциальные психологические проблемы, связанные с подробным коллективным рассмотрением программы каждого разработчика. Многие сравнивают этот способ с обсуждением в так называемых «группах мозгового штурма», которые были популярны в 60-х гг. Что касается коллективного обсуждения программ, то здесь существует опасность, что чрезмерно резкая критика со стороны группы может отрицательно сказаться на инициативности тех программистов, кто не обладает достаточной уверенностью в своих способностях.

7.7. Автоматизированные способы тестирования

Как мы видели в предыдущем разделе, многие теоретики вычислительной науки начинают довольно четко представлять себе, что наилучший способ устранить ошибки в программе — это прежде всего не допускать их проникновения в программу. Некоторые даже замечают, что ряд распространенных средств тестирования существует только для компенсации несовершенства применяемых методов программирования, которые со временем будут рассматриваться как устаревшие. Если бы только это было так!

К сожалению, кажется маловероятным, что мы достигнем этого идеала — программы, свободной от ошибок; вследствие этого мы будем вынуждены полагаться на классические способы и средства тестирования еще по меньшей мере несколько ближайших лет, а скорее еще долгие-долгие годы. В то же время *возможно* усовершенствование методики, которую применяет программист при формировании тестовых данных, при их вводе в программу и при контроле результатов тестового прогона,— это и есть тема настоящего раздела.

Основная мысль этого раздела заключается в том, что максимально возможную долю операций по тестированию следует выполнять без участия человека. Быть может, наиболее веским аргументом в пользу такого автоматического тестирования служит то, что *ручной* контроль ошибок сам по себе является потенциальным источником ошибок. Основная проблема состоит в том, что ошибочные результаты тестирования программист вполне может не заметить, просматривая бесконечные стопы распечаток результатов; возможно также, что какие-то результаты будут интерпретированы как ошибочные, в то время как фактически они правильны. В тестировании зачастую возникает и такая проблема: программист обычно психо-

логически склонен оценивать свои результаты как правильные; эту проблему мы обсудим ниже.

Существует еще другая проблема, которую не следует упускать из вида: ручноетестированиеосуществляетсязначительномедленнее и более утомительно, чем автоматизированные способы, о которых пойдет речь ниже. В связи с этим *объем* тестирования часто оказывается недостаточным; программист вручную формирует какие-то входные тестовые данные и тестовые файлы, вручную анализирует результаты теста, и через некоторое время эта работа надоедает и утомляет. Обычное следствие такой ситуации — неполностью проверенная программа.

Мы предлагаем здесь вместо ручного тестирования использовать некоторые виды программных подсистем, с помощью которых процесс тестирования в значительной мере автоматизируется и механизируется. Некоторые такие конкретные подсистемы рассматриваются ниже.

7.7.1. Автоматизированная генерация тестовых данных

Главная цель этого способа — обеспечить формирование больших объемов тестовых данных существенно механическим путем. Перед подробным обсуждением этого способа следует особо отметить следующее: *объем — не единственный критерий хорошего тестирования; вдействительностивбольшомобъемекроетсяпотенциальнаяопасность.* Мы заинтересованы в том, чтобы иметь *минимальный* объем тестовых данных, на котором можно будет адекватно проверить работу нашей программы; к сожалению, даже такой минимальный объем нередко оказывается за пределами возможностей ручного формирования.

По этой причине необходим генератор тестовых данных (обычно обозначаемый сокращением ГТД), позволяющий задавать форматы и диапазоны допустимых входных кодов, на основании которых он генерировал бы большие объемы таких данных. ГТД обычно может генерировать колоссальное количество *возможных* тестовых входных кодов, поэтому зачастую требуется тем или иным способом выделять из них приемлемую «выборку»; это чаще всего делается путем перевода ГТД в режим генерации *случайных* тест-данных (в пределах некоторого диапазона значений) или путем выбора тест-данных, распределенных с равными интервалами по всему диапазону возможных значений.

Одна из самых распространенных проблем в этой области состоит в том, что поведение программы часто зависит от *последовательности* подачи тестовых входных кодов. Вид отдельно взятого тестового входного кода может быть относительно простым, однако взаимосвязи между различными тестовыми ситуациями могут оказаться исключительносложными.Аналогично некоторыепрограммы

требуют *нескольких* различных входных параметров, и формирование рационального *набора* тест-данных, или тест-примеров, может представить довольно сложную задачу. Предположим, например, что программа производит относительно сложные вычисления, когда в нее поступают 50 различных входных параметров. Взаимосвязи между каждыми из 50 входных параметров являются, вероятнее всего, достаточно сложными, поэтому может оказаться целесообразным применить ГТД, генерирующий наборы из 50 случайных чисел.

Аналогичная ситуация имеет место при формировании тестовых *файлов;* чтобы как следует проверить большую программу, обычно требуется нетривиальный набор записей и файлов. Формировать такие тестовые файлы вручную часто затруднительно, а создать «живой» тестовый файл по различным причинам может быть невозможно. В то же время один из самых распространенных способов формирования тестового файла заключается в том, чтобы выбрать небольшое подмножество записей из *существующгго* файла (так часто поступают при обработке экономических данных, где разрабатываемая новая система должна заменить существующую систему ручной или машинной обработки данных).

Таблица 7.1

НЕКОТОРЫЕ РАСПРОСТРАНЕННЫЕ ПРОГРАММЫ-ГЕНЕРАТОРЫ ТЕСТОВЫХ ДАННЫХ

Название программы	Поста вщик
PRO/TEST	Synergetics Corporation One Carfield Circle Burlington, Massachusetts 01803
DATAMACS	Management and Computer Services, Inc. 104 Park Towne Place East Philadelphia, Pennsylvania 19130
IMI/TDQ-II	Information Management Inc. 447 Battery Street San Francisco, California 94111
TESTPAK	Computer Methods Corp. 470 Mamaroneck Avenue White Plains, New York 10605
TEST MASTER	Hoskyns Systems Research, Inc. 600 Third Avenue New York, New York 10016
FILEMAKE	Synergistic Software Systems P. O. Box 36097 Houston, Texas 77036
TESTGEN	Interface, Inc. 330 Municipal Court BuiWing Ann Arbor, Michigan 48108

Проблема генерации тестовых файлов аналогична проблеме генерации обычных тестовых входных кодов для программы: зачастую очень трудно строигь сложные базы данных. Приходится заботиться не только о генерации рационального числа перестановок и сочетаний индивидуальных записей (каждая из которых предположительно состоит из нескольких полей, способных принимать различные значения), но также о целесообразных *комбинациях* записей в файле и о возможных комбинациях файлов в большой базе данных.

Следует отметить, что ряд фирм, специализирующихся в области программного обеспечения, поставляет программы-генераторы тестовых данных на коммерческих основах. По очевидным причинам большинство этих программ ориентировано на Кобол и часто реализуется на машинах Систем 360 и 370 фирмы IBM. В табл. 7.1 перечисляются некоторые из наиболее распространенных генераторов тестовых данных.

7.7.2. Автоматизированный контроль результатов

Предпосылкой для создания программы автоматизированного контроля результатов служит основной принцип тестирования: *выполняя тест, вы всегда должны знать, какие результаты вы ожидаете получить.* Как упоминалось выше, программисты нередко склонны невольно находить оправдание или объяснение поведению своей программы (например: «Конечно, реально я не знал, какие результаты я получу, но не могу сказать, что эти результаты выглядят

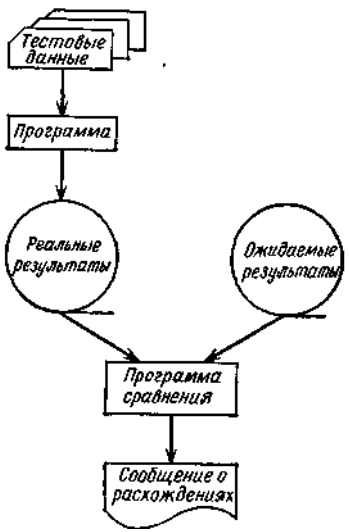


Рис. 7.5. Автоматизированный контроль результатов.

неправдоподобными»). Такой подход легко может привести к тому, что ошибки будут оставаться невыявленными.

Во многих случаях эта процедура верификации должна выполняться полуручным способом, т. е. заранее необходимо вручную определять, что для испытательного входного кода (тест-данных) А результатом теста должен быть X, для тест-данных В результатом должен быть Y и т. д. После этого мы легко можем написать программу, которая будет автоматически сравнивать *ожидаемые* (записанные в файле) результаты с *реальными* и выдавать на печать все расхождения. Эта процедура иллюстрируется рис. 7.5. Очевидно, что исключительно полезно иметь программу такого рода, если мы работаем с достаточно большими количествами тест-данных; в подобном случае мы явно избавляемся от колоссальных объемов трудоемких и сопряженных с ошибками работ по ручному контролю.

Иногда удается «механизировать» процедуру определения ожидаемых результатов; если это возможно, то процедуру проверки правильности реальных результатов также удается автоматизировать. Рассмотрим, например, проблему тестирования программы извлечения квадратного корня; для проверки работоспособности программы мы легко можем воспользоваться тем фактом, что

```
SQRT(X*X) = X
(SQRT(X))**2=X
```

для любого значенияX. Эта идея кажется довольно простой (если игнорировать проблему отбрасывания младших разрядов и округления, которая в приведенном примере может оказаться вовсе *не* тривиальной!) и довольно полезной для некоторых научных приложений, однако она, по-видимому, редко применима для систем, выполняющих экономические расчеты. В то же время она иногда может быть полезной для отдельных *модулей* системы экономического назначения (например, для контроля результатов работы модуля, в функции которого входит вычисление штрафа за превышение кредита в банке, или модуля, ответственного за вычисление налогов и удержаний в системе составления платежных ведомостей).

7.7.3. Автоматизированное управление тестированием

Программу тест-диспетчера, выполняющую эти функции, некоторые специалисты называют также «тест-монитором». Функции тест-диспетчера показаны на рис. 7.6. По существу тест-диспетчер действует как «главная программа». Он формирует входные тест-данные (быть может, принимая их от программы, которая способна генерировать их автоматически), подает их на испытуемую программу, получает результаты работы программы, распечатывает их и, быть может, помогает проверить правильность результатов. В связи сэтим использование тест-диспетчера может до некоторой степени соче-

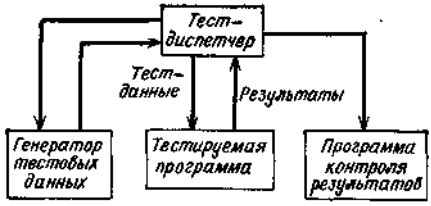


Рис. 7.6. Автоматизированное управление тестированием.

таться с генерацией тест-данных, описанной выше; такое использование может также сочетаться с идеей контроля правильности результатов теста. Важней всего то, что тест-диспетчер может быть сделан универсальным, чтобы все программисты, работающие над проектом, имели возможность применять один и тот же тест-диспетчер для тестирования индивидуальных модулей. Подобная возможность должна до некоторой степени сохраняться и для тестирования подсистем.

7.7.4. Автоматизация повторного тестирования

Как уже упоминалось в этой главе, исследования показали, что вероятность корректного внесения изменений в программу с первой попытки в общем случае не превышает 50%. Поэтому обычно целесообразно подвергать модифицированную программу полному повторному тестированию, даже если вносимое изменение кажется совершенно незначительным; особенно важно так поступать на этапе сопровождения программы, уже внедренной в эксплуатацию.

Основной принцип повторного тестирования иллюстрируется рис. 7.7. Вначале мы проводим тестовый прогон программы, проверяем правильность результатов и записываем их в файл. При любом повторном тестировании затем можно взять те же самые входные данные, произвести те же самые вычисления и получить, как можно надеяться, те же самые результаты. Результаты повторного тестирования можно затем сравнить с файлом результатов первого прогона и распечатать все расхождения.

В некоторых случаях мы можем внести значительные изменения в существующую программу (например, при исправлении серьезной ошибки). В подобных случаях следует ожидать, что результаты последующего тестового прогона могут *существенно* отличаться от результатов предыдущего. Чтобы при повторном тестировании программа не выдала слишком длинного сообщения о расхождениях, быть может, удастся снабдить ее управляющей информацией, показывающей, в каких полях результатов следует ожидать различий.

Более сложная проблема может возникать при тестировании программ реального времени. Ввиду различий во временных режи-

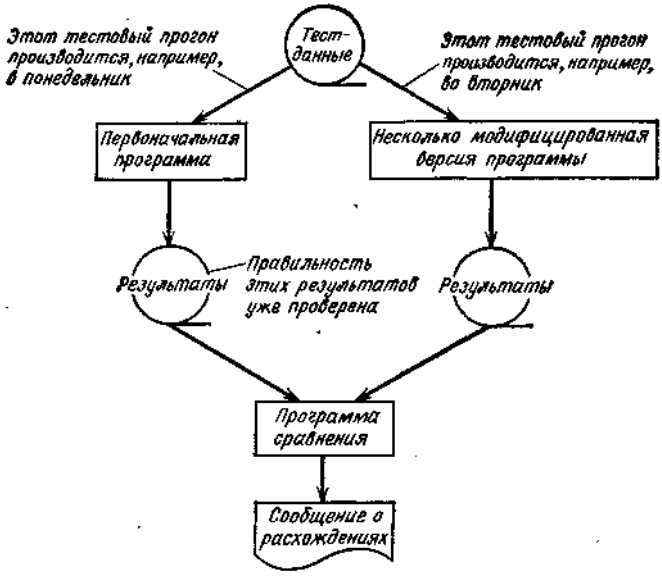


Рис. 7.7. Повторное тестирование.

мах при одной и той же последовательности входных тест-данных могут получаться два совершенно различных файла результатов. Однако может оказаться возможным подбор таких тест-данных, обработка которых не зависит от временных режимов. Может оказаться возможным и подбор такого набора входных тест-данных, чей совокупный результат не зависит от порядка обработки данных. Хотя ни один из этих способов нельзя считать достаточным для исчерпывающей проверки программы реального времени, они могут быть все же полезными при тестировании.

7.7.5. Автоматизированный контроль тестирования

Главная цель так называемых тест-мониторов состоит в том, чтобы определить, когда выполняется каждый оператор в программе, и зафиксировать тот факт, что он выполнен. После завершения программы тест-монитор печатает сообщение, показывающее, какие части программы проверены с помощью конкретных тест-данных. Сам по себе тест-монитор — это отдельная программа, которая может быть написана на любом удобном языке. Однако при этом обычно требуется, чтобы испытуемая программа выполнялась в режиме интерпретации; возможен и вариант, когда специальный дополнительный компилирующий блок (подобный отладочному компилятору языкаПЛ/1, поставляемомуфирмой IBM)вызывает включение в объектную программу для каждого оператора языка высокого уровня

дополнительных машинных команд, назначение которых—осуществлять условные передачи управления тест-монитору. Возможен еще один вариант, при котором препроцессор (программа предварительной обработки) включает соответствующие команды в исходную программу перед ее трансляцией или компиляцией.

После того как выполнение программы закончено, тест-монитор может выдать сообщение, содержащее информацию любого из следующих видов:

1. Перечень модулей, подвергнутых проверке (это особенно полезно для тестирования систем).
2. Точки входа в подпрограмму, подвергнутую проверке.
3. Перечень операторов IF, подвергнутых проверке.
4. Перечень всех операторов или команд в модуле, подвергнутых проверке.

Отметим, что тест-монитор — это эффективный контролирующий и управляющий инструмент; он дает возможность оценить в явном виде, насколько полная и основательная проверка работоспособности программы производится программистом с помощью выбранных им тестов. Кроме того, он может помочь сэкономить в дальнейшем машинное время, поскольку позволяет выбрать минимальный набор входных тест-данных, который будет проверять максимальный процент команд рабочей программы.

7.8. Другие методы тестирования

Существуют и другие достаточно разнообразные методы, применяемые в области тестирования вычислительных систем; хотя многие из них не получили широкого распространения, они все же заслуживают краткого упоминания. Некоторые из методов тестирования, описанных ниже, находятся пока еще в экспериментальной стадии и требуют дополнительного апробирования и (или) дальнейших исследований. Другие методы относятся только к очень крупным системам или системам с исключительно высокими требованиями к надежности программных средств (например, к категории, которая ранее в этой главе была определена как «гиперсложные» программы). Тем не менее некоторые из этих методов могут в конце концов оказаться полезными и практичными и для обычных систем.

7.8.1. Избыточное программирование

Основная идея здесь заключается в том, что для конкретного приложения независимо разрабатываются две (или более) программы. Каждая версия программы конструируется. кодируетсяитестируется отдельными группами программистов, которые предположительно не контактируют между собой, а только имеют одно и то же исходное техническое задание.

Этот принцип избыточного программирования может показаться довольно странным и дорогостоящим, однако его можно сравнить с принципом двукратного или многократного резервирования процессоров как способа обеспечения живучести системы при аппаратных отказах. Здесь можно провести следующую любопытную аналогию: в некоторых системах применяются *два переключаемых* ЦП, причем резервный ЦП используется для выполнения других работ, пока основной ЦП не выйдет из строя. В других системах применяются *сдвоенные* ЦП, причем оба ЦП выполняют одни и те же вычисления, результаты которых сравниваются. Ту же самую идею можно было бы распространить и на обеспечение избыточности программных средств. Несмотря на очевидную дороговизну, в ряде случаев такой подход может оказаться оправданным.

7.8.2. Стандартизация

Эта мысль довольно очевидна: проблемы, тестирования сводятся к минимуму, если применять подпрограммы и системы общего назначения. Если даже такие программы общего назначения содержат некоторые ошибки, их поведение обычно предсказуемо и известно, а это, как правило, лучше, чем иметь дело со специальной программой, содержащей неизвестные ошибки.

Практический недостаток программ общего назначения заключается в том, что *большие* программы общего применения (поставляемые фирмами-поставщиками системы управления базами данных, системы статистической обработки, системы приема и оформления заказов, системы составления платежных ведомостей и т. д.) часто содержат большое число ошибок и требуют постоянного сопровождения и развития. Небольшая специализированная программа, сдругой стороны, могла бы вначале содержать меньшее число ошибок. Иногда интересно сравнивать «живую» систему (систему, требующую постоянных модификаций и усовершенствований) и «статическую» систему (систему, изменения в которую вносятся только для устранения существующих ошибок); в общем случае мы вправе ожидать, что статическая система будет иметь гораздо меньше «аварий» в работе.

7.8.3. Моделирование

При разработке крупных и сложных вычислительных систем сейчас достаточно широко применяется *аналитическое* моделирование. Такое моделирование, связанное с созданием *моделей* вычислительной системы до ее практической реализации, может быть исключительно полезным для выявления ошибок быстродействия и емкости. В некоторых случаях оно может также помочь обнаружить и

грубые логические ошибки, однако подобное моделирование определенно не поможет находить ошибки кодирования.

Мы могли бы также упомянуть о существовании операционного моделирования (или имитирования), где имитационные входные сигналы и данные используются для тестирования *реальной* программы или системы. Моделирование такого типа довольно широко распространено в случае крупных систем, работающих в реальном масштабе времени, особенноввоенных и космических приложениях. Затраты на такое моделирование, безусловно, очень велики (зачастую они намного превышают стоимость конструирования и кодирования операционного программного обеспечения), однако есть ситуации, где не может быть другого эффективного способа тестирования (например, тестирование системы противовоздушной обороны, выполненной на базе ЭВМ).

7.8.4. Математическое моделирование надежности программ

Необходимость количественного определения надежности программных средств для сложных вычислительных систем (например, упомянутых выше систем противовоздушной обороны на базе ЭВМ) привела к тому, что в последнее время большое внимание стали уделять исследованиям методов математического *моделирования* надежности программ. На основании статистических данных, собираемых в процессе тестирования, надежностная модель должна позволить предсказать с достаточной вероятностью:

1. Число ошибок, остающихся в программе, которая вводится в эксплуатацию.
2. Наработку на отказ для *программных средств*.
3. Вероятность того, что программа (или программная система) будет успешно работать в течение заданного периода времени.

Принцип математического моделирования надежности программных средств был довольно успешно применен в одном космическом проекте НАСА и в одном или двух проектах министерства обороны США. Однако пока следует считать, что этот подход находится на ранней стадии своего развития. Многие из результатов математического моделирования базируются на некоторых упрощенных допущениях (к таким допущениям относится, например, то, что процесс исправления обнаруженной ошибки не вносит каких-либо новых ошибок). Тем не менее надо надеяться, что надежностное моделирование удастся через несколько лет применить и для «рядовых» систем; некоторые материалы по развитию этой области моделирования содержатся в работе Шумана [5] и в недавних записках проводившегося Институтом инженеров по электротехнике и радиоэлектронике симпозиума по надежности программных средств.

ЛИТЕРАТУРА

1. Randell B., Buxton J. N., Software Engineering Techniques, NATO Scientific Affairs Division, Brussels 39, Belgium, 1970.
2. Boehm B Software and Its Impact: A Quantitative Study, Datamation, May 1973.
3. Yourdan E., Design of On-Line Computer Systems, Prentice-Hall, Inc., 1972.
4. Yourdan E., Reliability of Real-Time Computer Systems, Modern Data, Jan.—June 1972.
5. Shooman N. L., Dickson J. C., Hesse J. L., Kientz A. C., QuantitativeAnalysis of Software Reliability, IEEE Symposium on Software Reliability, Jan. 1972.
6. Sackman H., Erickson W. J., Grant E. E., Exploratory Studies Comparing Online and Offline Programming Performance, Communications of the ACM, Jan. 1968, p. 3—11.
7. Boehm B., Some Information Processing Implications of Air Force Space Missions in the 1980's, Astronautics and Aeronautics, Jan. 1971, p. 42—50.
8. That Maintenance Iceberg, EDP Analyzer, Oct. 1972.
9. Baker F. T., Chief Programmer Team Approach to Production Programming, IBM Systems Journal, Jan. 1972.
10. Hetzel W. C. (ed.), Program Test Methods, Prentice-Hall, Inc., 1973.
11. Weinberg G., The Psychology of Computer Programming, D. Van Nostrand Co., Inc., 1971.

ВОПРОСЫ

1. Приведите краткое определение (а) модуля, (б) программы, (в) подсистемы и (г) системы.

2. Приведите определение термина «надежность программных средств». Имеет ли это понятие смысл применительно к проектам, над которыми вы обычно работаете?

3. Приведите определение (а) ошибки и (б) ляпа. Как вы думаете, согласятся другие программисты вашей организации с вашими определениями? Какая разница между ошибкой и ляпом? Согласятся ли с вами пользователи и руководство?

4. Приведите определение «тестирования». В чем цель тестирования машинной программы?

5. Приведите краткое определение (а) апробирования, (б) верификации и (в) аттестации. Приняты ли эти термины в вашей организации? Часто ли они используются?

6. Приведите определение термина «отладка». В чем различие между тестированием и отладкой? Не кажется ли вам, что многие программисты обычно путают эти два понятия? Существенно ли это?

7. Приведите краткое определение термина «сопровождение». В чем различие между разнообразными видами действий, которые обычно скрываются подобщим названием «сопровождение»?

8. Если вы применяете стандартную операционную систему, предоставляемую фирмой-поставщиком, попытайтесь собрать некоторые статистические данные относительно числа ошибок, выявленных в нескольких последних версиях системы. Эти данные могут быть получены от фирмы, групп пользователей или вашего отдела по эксплуатации вычислительных средств. Можете вы заметить ту или иную четкую тенденцию в изменении числа ошибок (т. е. растет ли оно, уменьшается или остается постоянным)? Разделите число ошибок на число команд операционной системы: не думаете ли вы, что полученный показатель «число ошибок на команду» значительно выше, чем для прикладных программ, с которыми вы обычно имеете дело?

9. Соберите некоторые статистические данные относительно количества времени, затраченного на тестирование при разработке нескольких проектов в вашей организации. Была ли когда-либо эта цифра меньше 30%? Если да, то не вызвали

ли такие программы много хлопот после того, как были переданы на сопровождение и эксплуатацию? Находите ли вы, что существуют заслуживающие внимания важные корреляционные связи, например:

а) Между временем тестирования и временем разработки?

б) Между временем тестирования и размером программ?

в) Между временем тестирования и квалификацией программистов?

г) Между временем тестирования и применяемым языком программирования? Считаете ли вы, что длительность периода тестирования меняется от проекта к проекту или остается относительно постоянной?

10. Какую часть бюджета вычислительного центра вашей организации составляют затраты на сопровождение? Знает ли об этом ваше руководство? Не думаете ли вы, что основную часть затрат на сопровождение можно было бы исключить благодаря более тщательному тестированию?

11. Приведите пример простой программы, определенной в подразд. 7.3.1. Типичны ли такие программы для вашей организации?

12. Приведите пример программы средней сложности, по определению подразд. 7.3.2. Типичны ли такие программы для вашей организации? Знаете ли вы о каких-либо проектах такого масштаба, которые не удалось довести и до конца и пришлось прекратить из-за трудностей тестирования?

13. Приведите пример сложной программы, определенной в подразд. 7.3.3. Типичны ли такие программы для вашей организации? Приведите пример сложной программы, которую пришлось прекратить из-за трудностей тестирования. Что, по вашему мнению, было основной причиной неудачи этого проекта: неправильное конструирование или неправильное тестирование программы? Как можно было бы решить эту проблему?

14. Приведите пример сверхсложной программы, определенной в подразд. 7.3.4. Разрабатывали ли в вашей организации подобные проекты? Каково отношение между числами удачных и неудачных проектов?

15. Приведите пример гиперсложной программы, определенной в подразд. 7.3.5. Участвовали ли вы когда-либо в разработке такого проекта? Известны ли вам удачные проекты подобного рода, т. е. системы, функциональные возможности которых соответствовали заданным требованиям, а разработка которых была выполнена в срок, в пределах бюджета и с достаточной степенью изящества?

16. Приведите пример логической ошибки, которую можно было бы выявить путем тестирования. Какую долю всех ошибок составляют, по вашему мнению, логические ошибки?

17. Приведите пример ошибки документации (согласно подразд. 7.4.2). Насколько серьезны, по вашему мнению, ошибки документации? Выявляются ли они с помощью процедур тестирования, применяемых в вашей организации?

18. Приведите пример ошибки перегрузки (согласно подразд. 7.4.3). Считаете ли вы типичными ошибки такого рода? Как можно тестированием выявить подобную ошибку?

19. Приведите пример временной ошибки (согласно подразд. 7.4.4). Существуют ли подобные ошибки в обычных прикладных программах пакетной обработки? Какую часть ошибок в программе реального времени могут составлять, по вашему мнению, временные ошибки? Как можно выявить временные ошибки тестированием?

20. Приведите пример ошибок быстродействия и емкости (согласно подразд. 7.4.5). Могут ли такие ошибки встречаться в прикладных программах пакетной обработки или они бывают только в системных программах? Как можно выявить эти ошибки тестированием. Проводит ли кто-нибудь в вашей организации тестирование с целью выявления ошибок перегрузки?

21. Приведите пример ошибки возврата и восстановления (согласно подразд. 7.4.6). Как можно выявить подобные ошибки тестированием? Считаете ли вы, что это важная проблема?

22. Приведите пример аппаратных ошибок и ошибок системных программных средств (согласно подразд. 7.4.7). Считаете ли вы, что целесообразно проводить

тестирование для выявления таких ошибок? Сколько времени следует затрачивать на такое тестирование? Каким образом его следует проводить?

23. Приведите пример ошибки несоблюдения стандартов (согласно подразд. 7.4.8). Считаете ли вы, что важно проверятьлрограммы на отсутствие таких ошибок? Делается ли это в вашей организации? Каким образом следует это делать? Следует ли производить проверку вручную?

24. Почему важно тестировать программу поэтапно? Считаете ли вы, что это действительно необходимо для простых программ? Как можно определить, следует ли проводить официальные испытания программы?

25. Приведите определение тестирования модулей. Назовите некоторые синонимы этого термина. Почему так важно, чтобы тестирование модулей осуществлялось полно и всесторонне? Каким образом можно удостовериться, что это условие выполнено?

26. Имеются ли в вашей организации статистические данные, показывающие, какое среднее число выходов на машину (т. е. попыток тестирования) требуется программисту для тестирования индивидуальных модулей? Если подобной статистики не существует, не считаете ли вы, что программисты будут возражать против попыток сбора таких статистических данных? И если так, то почему?

27. Приведитеопределениетестирования подсистем (согласно подразд. 7.5.2). Проходят ли проекты в вашей организации официальный этап тестирования подсистем? Считаете ли вы, что требуется более одного такого этапа? Каким образом решается этот вопрос?

28. Приведите определение тестирования системы. Когда можно сказать, что выполнен достаточный объем предварительного тестирования и разрешается приступать к тестированию системы? По каким признакам мы судим о том, когда следует прекращать тестирование системы? Сколько времени в вашем проектеобычно отводится системному тестированию?

29. Приведите определение приемо-сдаточных испытаний. Применяются ли в вашей организации какие-либо синонимы этого термина? Кто руководит приемо-сдаточными испытаниями? Кто разрабатывает тест-данные для этих испытаний? Кто решает, когда следует закончить подобные испытания? Для всех ли проектов в вашей организации требуется проведение официальных приемо-сдаточных испытаний?

30. Приведите определение нисходящего тестирования. Чем оно отличается от восходящего тестирования? В чем преимущества нисходящего тестирования?

31. Приведите пример затруднений, которые могут возникнуть по недосмотру программиста ввиду нечеткости синтаксиса таких языков программирования, как Фортран и ПЛ/1. Встречались ли когда-либо подобныезатруднения в ваших собственных программах? Что можно сделать, чтобы устранить подобные проблемы?

32. В подразд. 7.6.1 описан ряд «неоднозначных» ситуаций, встречающихся при программировании. Приведите пример подобной ситуации для вашего собственного языка программирования. Можете ли вы предложить какой-нибудь возможный способ выхода из такой ситуации?

33. Принято ли в вашей организации позволять разрабатывать проекты групповым методом, о котором шла речь в подразд. 7.6.4? Приводит ли это к существенному сокращению длительности тестирования? Не вызывает ли это каких-либо других проблем?

34. Какова главная цель применения автоматизированных способов тестирования, рассмотренных в подразд. 7.7? Считаете ли вы эту цель оправданной? В чем заключаются опасности ручного тестирования?

35. Приведите пример программы, тестирование которой можно было бы упростить при наличии генератора тестовых данных, описанного в подразд. 7.7.1. Применяют ли в вашей организации подобный генера»ор? Всем ли программистам он требуется?

36. Приведите пример программы, тестирование которой нельзя было бы су-

щественно упростить при наличии генератора тест-данных. Считаете ли вы такие программы распространенными?

37. В порядке научного исследования проанализируйте генераторы тест-данных, перечисленные в табл. 7.1. В чем их достоинства и недостатки? Какой из них вы могли бы рекомендовать для своей организации?

38. Приведите пример программы, тестирование которой можно было бы упростить благодаря применению подсистемы автоматизированного контроля результатов, о чем шла речь в подразд. 7.7.2. Какова основная задача подобной подсистемы? Считаете ли вы, что ее можно было бы применить в большинстве программ, разрабатываемых вашей организацией?

39. Приведите пример программы, которую нельзя было бы легко проверить с помощью подсистемы автоматизированного контроля результатов. Считаетели вы, что это распространенная ситуация?

40. Приведите пример программы, тестирование которой можно было бы упростить благодаря применению тест-диспетчера, описанного в подразд. 7.7.3. Применяют ли в вашей организации подобный тест-диспетчер? Всем ли программистам он требуется? Какое значение имеет применение тест-диспетчера при нисходящем или восходящем тестировании? Считаете ли вы, что это важно?

41. В чем цель способа повторного тестирования, описанного в подразд. 7.7.4? Считаете ли вы, что это действенный способ? Считаете ли вы, что можно было бы разработать универсальную подсистему повторного тестирования для применения всеми программистами вашей организации? Если нет, то почему?

42. В чем назначение автоматизированного тест-монитора, описанного в подразд. 7.7.5? Считаете ли вы его применение оправданным? Считаете ли вы, что подобную подсистему можно было бы разработать для применения всеми программистами вашей организации?

43. В чемцель избыточного программирования? Разрабатываются ли в вашей организации таким способом какие-либо проекты? Считаете ли вы, что такой подход к программированию мог бы стать экономически оправданным?

ГЛАВА 8

ПРИНЦИПЫ И СПОСОБЫ ОТЛАДКИ

8.0. Введение

Как мы указывали в предыдущей главе, *тестирование* — это специально организованная процедура, которая предназначена для выявления наличия ошибок в машинной программе. Это трудоемкая, сложная и часто неблагодарная задача; немногие программисты получают от нее удовлетворение и еще меньше число тех, которые обладают природными способностями для ее выполнения.

В данной главе обсуждается *отладка* — тот вид работы при программировании, который тесно соприкасается с тестированием и который с ним часто путают. Отладка — это искусство: искусство выявления природы ошибки и локализации ошибки после того, как установлено ее наличие. Хотя специально организованные процедуры и «механические» средства (такие, как дампинги памяти) занимают существенное место в процессе отладки, хороший программист обычно находит сложнуюошибкуинтуитивноблагодаря «божественному вдохновению». Как и тестирование, отладка — это не тот вид работы, который привлекателен для среднего программиста; и в действительности этот процесс обычно считают тем аспектом разработки программы, где чаще всего происходит срыв планов и наблюдается больше всего нервотрепки. Большинство программистов не обладают необходимым талантом для отладки; как правило, они стараются одолеть программную ошибку с помощью больших дампов, трассировочных таблиц и прочих методов, опирающихся на «грубую силу».

В данной главе мы ставим перед собой двойную цель. Во-первых, мы рассмотрим некоторые методологические и стратегические принципы, применяемые теми, кто подходит к отладке, как к искусству. Ряд методологий будет предусматривать использование ЭВМ, но многие из них — нет. Кроме того, мы рассмотрим ряд подсистем (пакетов программных средств) отладки, которые традиционно применяются для локализации ошибок в машинных программах. После объяснения некоторых классических приемов отладки (таких, как дампы, трассировки и т. д.) мы сосредоточим свое внимание на под-

системе динамической отладки с помощью терминала, называемой ДДТ (Dynamic Debugging Technique).

Следует подчеркнуть, что хорошо написанная программа уменьшит необходимость в применении тех способов отладки, которые описаны в данной главе. В гл. 1, например, мы обсуждали ряд предложений, направленных на уменьшение сложности отладки; здесь уместно было бы освежить в памяти содержание разд. 1.2. В гл. 3 и 4 предлагался ряд способов написания модульных программ; было показано, что, помимо всего прочего, модульную программу легче отлаживать, чем немодульную. В гл. 5 отмечалось, что простую, легкую для понимания программу отладить значительно проще, чем программу, в которой программист пошел по пути применения заумных, усложненных приемов программирования. Наконец, в гл. 6 обсуждалась идея программирования с защитой от ошибок, т. е. идея введения в программу достаточного количества средств контроля для выявления всех ошибок, за исключением наиболее тонких и замаскированных.

Даже если следовать всем этим рекомендациям, ошибки будут неизбежно встречаться и в большинстве самым тщательным образом написанных программ; ошибаются все программисты. По-видимому, максимум, на что мы можем надеяться,— это то, что наши ошибки можно будет легко находить. Именно учитывая все это, мы излагаем в последующих разделах соответствующие способы отладки.

8.1. Методологические и стратегические принципы отладки

Для многих начинающих программистов отладка отождествляется с дампингами оперативной памяти, контрольными выводами данных («фотографиями») и получением трассировочных таблиц; в действительности же для многих машин дампинги памяти и трассировочные таблицы настолько неудобны и невразумительны, что программист прибегает к ним только как крайней мере. Не один программист, вероятно, говорил себе: «Если бы только знать, как правильно применять этот пакет для отладки программ Кобола, у меня никогда бы не было никаких проблем с отладкой...» В этих его иллюзиях нет ничего странного, если прочитать главу, посвященную отладке, в руководстве по Коболу.

Более опытный программист (а также некоторые начинающие программисты, обладающие природным даром к такой работе) знает, что существуют многие другие возможности для отладки программы, помимо дампов памяти. Поиск программной ошибки во многом схож с выслеживанием убийцы в детективе Агаты Кристи: там, где массовые облавы и прочие методы грубой силы, применяемые местной полицией, терпят неудачу, герой романа, Эркюль Пуаро, преуспевает благодаря «маленьким клеткам своего серого вещества».

То же самое обычно справедливо и для программ. Простые, очевидные ошибки можно найти с помощью контрольных выводов данных и трассировок, *но обычно их можно обнаружить так же легко (причем с очень существенной экономией машинного времени), затратив немного времени на целенаправленные размышления*. Поистине сложные ошибки, например ошибки, встречающиеся в очень больших системах пакетной обработки, а также в системах коллективного пользования и реального времени, часто удается найти исключительно с помощью некоторых способов, предложенных в данном разделе и не связанных с применением ЭВМ. Перефразируя характеристику подхода, используемого прославленными сыщиками из детективов, можно сказать, что мы должны понять «психологию» ошибки и психологиюпрограммиста, которыйвнесэтуошибкувпрограмму.Поэтомунамкажется весьма удачным замечание Хопкинса: «Программная ошибка — это наша ошибка, и если мы будем чуть больше знать о видах ошибок, которые нам свойственны, мы сможем чуть легче обнаруживать и исправлять эти ошибки (и, возможно, даже предотвращать их появление в будущем!)».

Имея это в виду, мы обсудим ряд стратегических принципов, облегчающих поиск ошибок в машинной программе.

8.1.1. Пишите небольшие модули

Одним из самых простых путей облегчения последующей отладки программы является разделение ее при разработке на небольшие модули. У нас нет детальных цифровых данных, чтобы обосновать это утверждение, однако, по-видимому, количество и сложность ошибок экспоненциально возрастают в зависимости от количества команд в модуле; таким образом, если модули будут небольшого размера, мы облегчим поиск ошибок.

Очевидно, что при уменьшении размеров модулей (или подпрограмм, или параграфов программ на Коболе и т. д.) мы, как правило, увеличим количество модулей в программе. По мере увеличения количества модулей и усложнения взаимосвязей между ними можно ожидать появления большего числа ошибок в программе. Если, однако, программа разрабатывается по иерархическому принципу, описанному в гл. 2, то число взаимосвязей между модулями можно будет удержать в приемлемых границах.

Если для выполнения своих функций программа требует 100 команд, то ее можно было бы написать в виде одного модуля (или главной программы) из 100 операторов; другой крайний случай — ее можно было бы написать в виде 100 модулей, каждый из которых содержит один оператор (с некоторыми дополнительными операторами для связи «подпрограмм» и т. д.). Оба этих крайних подхода неудовлетворительны, причем трудно сказать, в каком случае программа оказалась бы сложнее в отладке. Если число ошибок дейст-

вительно находится в экспоненциальной зависимости от числа команд в модуле и числа модулей в программе, то, быть может, нам удалось бы свести число ошибок кминимуму, если бы разделить программу на десять модулей по десять операторов в каждом. На практике, по-видимому, наилучшим решением является уменьшение размера модуля (с соответствующим увеличением числа модулей в программе) до такой степени, при которой дальнейшее его сокращение становится бессмысленным.

8.1.2. Постарайтесь определить природу ошибки

В гл. 7 обсуждались некоторые типы ошибок, которые могут встретиться в программе. Если тестирование проводится достаточно тщательно, то, возможно, удастся выявить общую причину неработоспособности программы еще до того, как программист начнет поиск с целью выявления конкретной природы ошибки. Поэтому, прежде чем начинать поиск ошибки, вы должны попытаться определить, ищете ли вы ошибку аппаратуры, ошибку операционной системы, ошибку компилятора или же ошибку в собственной программе. Если ошибка, по-видимому, все же находится в вашей собственной программе, то возникает вопрос, какой модуль (или группа модулей), на ваш взгляд, окажется виновным?

В качестве примера рассмотрим образец итоговой выдачи, показанный на рис. 8.1; очевидно, что все промежуточные суммы для каждого отдельного территориального участка сбыта товаров неверны. Поскольку все цифры для отдельных торговых агентов кажутся разумными (хотя нет гарантии, что они верны), можно предположить, что ошибка должна возникать в той части программы, где производится подсчет отдельных итоговых сумм, либо, быть может, в подпрограмме, печатающей результаты. Поскольку маловероятно, что неверные числа появились вследствие аппаратной ошибки либо по некоторой другой непонятной причине, эти причины, видимо, не следует принимать во внимание, пока проведение анализа логического хода программы не зачеркнет эти предположения.

Предположим теперь, что неправильны итоговые суммы лишь для *одного* участка сбыта товаров, а итоговые суммы остальных участков, как это показано на рис. 8.2, верны. В этом случае также вероятно, что ошибка была вызвана некорректностью логики программы; а понять это программист может, лишь изучив отдельные сообщения о сбыте товаров по данному участку. Однако имеет смысл, прежде чем всерьез начинать поиск ошибки (особенно если предполагается, что это сложная и тонкая ошибка), задать себе несколько других немаловажных вопросов:

1. Не случилось ли что-нибудь с быстродействующим печатающим устройством?

Лист 001 Объем продаж по участку 001

Торговый агент	Объем продаж текущего месяца	Нарастающий итог для данного года (до текущего момента)	В этом же месяце прошлого года
Браун	2000	6387	1998
Каннинг	1720	5291	840
Деннис	1909	6066	1837
Янг	1675	4280	1500
Циммерман	2100	8200	2025
Итого	600	423	68234

Лист 002 Объем продаж по участку 002

Торговый агент	Объем продаж текущего месяца	Нарастающий итог для данного года (до текущего момента)	В этом же месяце прошлого года
Андерсон	2288	7254	1904
Беннет	1803	6377	1776
Уотсон	1313	6281	1505
Уилсон	1838	5530	1700
Итого	209	156	75236

2. Не могла ли данная ситуация возникнуть из-за нестандартной ошибки во входных перфокартах или в главном файле?

3. Заметил ли оператор ЭВМ что-либо необычное во время выполнения программы? Замечал ли он когда-нибудь что-либо подобное?

4. Выполняется ли программа в тех же самых условиях, как и во время последнего прогона? То есть не была ли после завершения этого последнего прогона произведена перекомпиляция исходной программы или генерация новой операционной системы? Не могло бы это объяснить ситуацию?

5. Надежно ли работала последнее время еппаратура? Возможно ли, что произошел сбой при выполнении арифметических операций?

Лист 001 Объем продаж по участку 001

Торговый агент	Объем продаж текущего месяца	Нарастающий итог для данного года (до текущего момента)	В этом же месяце прошлого года
Браун	2000	6387	1998
Каннинг	1720	5291	840
Деннис	1909	6066	1837
•	•		•
Янг	1675	4280	1500
Циммерман	2100	8200	2025
Итого	62109	157613	57846

Эти цифры правильны.

Лист 046 Объем продаж по участку 042

Торговый агент	Объем продаж текущего месяца	Нарастающий итог для данного года (до текущего момента)	В этом же месяце прошлого года
Абраме	1735	5319	1409
Эдварде	2066	5805	1892
•	•	•	•
Смит	1543	4212	1680
Томпсон	2248	7227	2071
Итого	413	63	153814

Эти цифры неправильны.

Рис. 8.2. Более сложная программная ошибка.

В большой программе (или системе программирования) не всегда просто определить природу ошибки; как мы уже отмечали в гл. 6, ошибка может распространиться подобно раковой опухоли. К тому жемногие из наиболее сложных ошибок возникают, по-видимому, на стыке нескольких частей системы. Во многих системах коллективного пользования, например, у программиста часто бывает крайне сложная ситуация при определении того, была ли ошибка вызвана пользователем, терминалом, линией связи, мультиплексором, аппаратурой ЭВМ, математическим обеспечением поставщика (например, операционной системой) или его собственной программой.

Даже в такой сложной ситуации можно достичь некоторого прогресса путем *исключения из рассмотрения маловероятных источ-*

никое ошибки. Если проблема была вызвана ошибкой программы, то какие части программы могут быть исключены из рассмотрения? Какие части аппаратуры можно не учитывать в качестве вероятных источников неприятностей? За исключением наиболее сложных ситуаций, обычно удается сузить область поиска до двух или трех вероятных кандидатов, а затем уже продолжать последовательный анализ и поиск ошибки.

Это один из многих аспектов процесса отладки, которую, собственно говоря, можно считать *искусством;* некоторые программисты в этом преуспевают, а некоторым она не дается. Тем не менее это также способ, который можно освоить на практике. Вы можете вовсе не обладать шестым чувством, которое позволяет иногда буквально чуять ошибку, но вы должны быть в состоянии повысить свою квалификацию, подняв ваше мастерство над уровнем «беспорядочного поиска», а ведь именно к этому сводится отладка программ у многих программистов.

8.1.3. Проверьте, является ли ошибка повторяющейся и устойчивой

Если не предполагается, что ошибка достаточно проста, целесообразно проверить, является ли она устойчивой и повторяющейся. В случае ошибки, показанной на рис. 8.1, быть может, не обязательно повторять выполнение программы: природа ошибки говорит о том, что она, по-видимому, довольно устойчива. Однако ошибка, показанная на рис. 8.2, далеко не так очевидна — итоговые объемы продажи неправильны лишь для одного участка. Если неправильный результат — это следствие программной ошибки и если ее можно легко найти, тем лучше. В то же время, если тщательная проверка программы не приводит к обнаружению ошибки, то, быть может, имеет смысл вторично выполнить программу, чтобы убедиться в том, что ошибка действительно существует.

Имеется несколько возможных причин, способствующих возникновению *неповторяющейся* ошибки.

1. Оператор ЭВМ мог сделать что-то, что привело к неправильной работе машины и (или) программы. Он мог, например, толкнуть устройство ввода с перфокарт, быстродействующее печатающее устройство или накопитель на магнитной ленте, вызвав тем самым сбой (который мог остаться не обнаруженным системой или не замеченным оператором). Аналогично программа могла потребовать какого-то указания от оператора, а он мог напечатать на пультовом телетайпе неправильное сообщение (такая ошибка должна обнаруживаться без повторного прогона программы!).

2. Подозреваемая программа может выполняться в режиме реального времени и оказываться в условиях, которые зачастую назы-

ваются «временной ошибкой», т. е. ошибка может произойти потому, что две программы одновременно пытались обновить одну и ту же запись базы данных, или потому, что одна программа неожиданно прервала другую программу и испортила ее рабочую память. Даже если сама программа не является программой реального времени, то она обычно выполняется под управлением мультипрограммной операционной системы фирмы-поставщика; не исключено, что временная ошибка в этой операционной системе проявляется как кажущаяся ошибка ни в чем не повинной прикладной программы.

3. Аппаратные средства ЭВМ могут работать со сбоями. Незаземленный кабель или перегретый компонент могут привести к неправильному выполнению одной из команд машинного языка; необнаруженная ошибка ввода-вывода или ошибка канала могут испортить часть записи базы данных при считывании с магнитной ленты или диска. Ошибки временной диаграммы работы аппаратуры, которые имеют в общем такую же природу, как и рассмотренные выше временные ошибки *программных средств,* могут вызвать потерю запроса на прерывание или порчу данных.

Если программа очень велика или очень сложна, может быть очень трудно (или дорого) выполнить ее повторно до того момента, когда в ней встречается ошибка. Это дает нам еще один аргумент в пользу написания малых модулей: в идеале хорошо было бы иметь возможность повторять только тот модуль, в котором встречается ошибка. Хотя это и не всегда возможно, использование модульного принципа должно способствовать исключению повторного выполнения всей программы.

Если ошибка является неповторяющейся, то не всегда ясно, что делать программисту. Если предположить, что ошибка произошла не по вине человека (например, оператора), то можно рекомендовать следующее:

1. Повторите прогон программы и, в случае получения правильных результатов, забудьте о том, что ошибка имела место. Хотя такой подход нельзя рекомендовать как правило, он в некоторых случаях может оказаться единственным практически возможным. В действительности вы никогда не должны забывать ошибки такого типа, поскольку когда-нибудь они могут возникнуть снова. В ваших собственных интересах вам следует фиксировать такие таинственные, неповторяющиеся и необъяснимые ошибки.

2. Выполните те или иные диагностические тесты для проверки аппаратуры и надейтесь, что при этом удастся обнаружить какую-либо аппаратную неисправность. Если ситуация была вызвана сбоем аппаратуры, то диагностические тесты, скорее всего, не покажут ничего плохого. Более эффективный способ — это прогон диагностических тестов в комплексе с операционной системой, т. е. так, как если бы они были программой пользователя, выполняемой в мультипрограммных условиях.

3. Считайте, что ошибка находится в ваших собственных программных средствах и начинайте ее поиск.

4. Считайте, что ошибка произошла по оплошности оператора и возложите вину на него; он, по-видимому, находится на достаточно низком уровне иерархии в вашей организации, так что не сможет как следует защищаться (однако не ждите каких-либо любезностей от этого оператора в течение следующих шести месяцев!).

5. Если все это не даст результатов, попытайтесь возложить вину на поставщика — либо на его аппаратуру, либо на программное обеспечение. Быть может, это вовсе не его вина, однако не будет никакого вреда, если вы таким образом принудите его защищаться,— будьте только уверены, что ошибка не в вашей собственной программе!

8.1.4. Программист, познайсебя

Наибольше трудности с отладкой своих программ чаще всего испытывают те программисты, которые сваливают вину за *все свои* проблемы на аппаратуру, на операционную систему, на транслятор или на чужую программу. В моей практике был удивительный период совместной работы с одним программистом, который, как только у него возникали какие-либо трудности с программой, жаловался, что вычислительная машина заражена антисемитизмом.

Вы должны считать, что почти все ошибки, с которыми вам придется столкнуться в своей работе по программированию — это *ваши* ошибки, *ваши* оплошности. Осмыслив виды ошибок, которые вы делаете чаще всего, вы сможете, как правило, легче их находить. Некоторые программисты, например, замечают за собой, что они многократно делают одни и те же ошибки при перфорировании (да, некоторые программисты сами подготавливают перфокарты собственных программ!). Другие замечают, что они постоянно неправильно используют некоторую конкретную команду языка ассемблера или оператор Кобола. Некоторые также замечают, что они забывают запоминать и восстанавливать содержимое регистров при переходе на подпрограмму на языке ассемблера и при выходе из этой подпрограммы. А программисты, работающие на Фортране, часто обнаруживают, что они имеют обыкновение применять переменные I, J и K в нескольких различных местах программы, так что одна часть программы портит значение, записанное другой частью той же программы.

Один из способов познания собственных ошибок заключается в том, чтобы вести дневник своих работ по отладке. После того как вы разработаете несколько программ, этот дневник покажет вам характер ваших ошибок, что позволит вам более успешно работать в дальнейшем.

8.1.5. Будьте скрупулезны, методичны и логичны в поиске ошибок

Мы уже упоминали, что некоторые программисты испытывают затруднения при поиске более тонких ошибок в своих программах; другие ловко прослеживают ошибку и находят ее быстро и легко. Одна из причин этого заключается в том, что программисты более высокой квалификации при поиске ошибок применяют *методичный, систематический* подход. Менее квалифицированный программист, сдругой стороны, обычно ограничивается случайным зондированием своей программы, все больше и больше теряясь и впадая в панику с каждой неудачной попыткой найти ошибку.

Одним из примеров систематического подхода к отладке является метод «дихотомического поиска ошибок», который иллюстрирует рис. 8.3. Допустим, имеется программа, о которой известно, что в ней где-то скрывается ошибка. Мы можем проанализировать- состояние программы после того, как ее выполнение дойдет до средней точки. Если что-то уже окажется неверным (и если природу ошибки нельзя сразу же определить при анализе программы в этой точке), то мы можем повторно выполнить программу и остановить ее в середине первой половины прогона и т. д. Применение такого метода значительно упрощается при наличии подсистемы отладки под названием ДДТ (см. разд. 8.3).

В равной степени приемлем любой другой систематический подход; главное, чего мы хотим избежать,— это бессистемности в отладке. Каждый дамп оперативной памяти, каждая трассировочная таблица, каждая процедура анализа должны давать дополнительную информацию о природе и местонахождении ошибки,— либо исключая определенные части программы, которые, скорее всего, невинов-

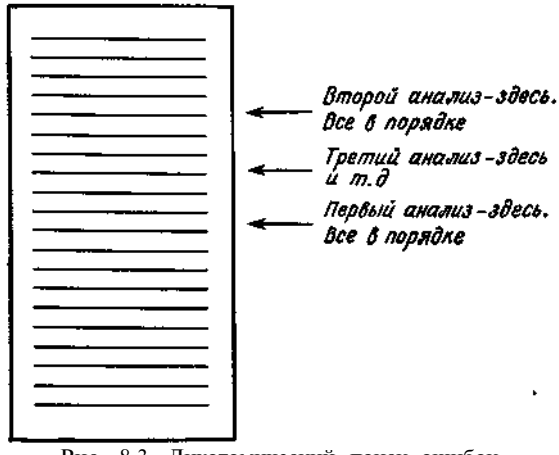


Рис. 8.3. Дихотомический поиск ошибок.

ны, либо дополнительно подтверждая первоначальное подозрение о местонахождении ошибки.

Все это может показаться противоречащим нашим предыдущим рассуждениям о шестом чувстве, или интуиции, которая помогает некоторым программистам искать свои ошибки; однако интуиция и систематичность не обязательно исключают друг друга — в действительности одно должно дополнять другое. Нет ничего плохого в том, что программист, который сталкивается с ошибкой в большой и сложной программе, говорит себе: «Мое шестое чувство подсказывает мне, что ошибка, вероятнее всего, содержится в подпрограмме АВС». Если, однако, это первое предположение оказывается неправильным, то программист не должен делать второе интуитивное предположение совершенно произвольно; вместо этого ему следует воспользоваться только что полученной информацией для определения своего следующего шага. Например, он мог бы сказать себе: «Если бы ошибка была в начале программы, то она проявилась бы в неработоспособности подпрограммы АВС; поскольку анализ работы АВС показал, что эта подпрограмма все делает правильно, то весьма вероятно, что ошибка находится где-то *после* АВС».

В случае небольшой программы или простой, очевидной ошибки все эти предосторожности не обязательны. С другой стороны, поиск некоторых ошибок в крупной и сложной программе занимает длительное время — иногда от трех до шести *месяцев*. Для подобной ситуации большое значение имеет систематичность и методическая обоснованность отладки. Если период отладки сильно затягивается, неплохо даже завести журнал, где бы записывались все положительные или отрицательные результаты, полученные в текущих работах, все проверенные подпрограммы и выводы, сделанные на основании промежуточных результатов. Фактически, если видно, что отладка, скорее всего, займет стольдлительное время (во многих проектах по программированию она достигает 50% общего срока выполнения работ), с помощью такого журнала можно будет также убедить вашего руководителя, что вы не бездельничаете.

8.1.6. Прежде всего проверяйте самые простые предположения

Этим советом я обязан своему студенту, мистеру Ли; интересно, что точно такой же совет дается во многих детективах: вначале проверяйте очевидные вещи.

В случае отладки программы, познакомившись с краткой характеристикой задачи, обычно можно составить список подпрограмм, которые в принципе должны вызывать подозрение при ошибках. Так, услышав, что ваша программа выдает результаты неправильного вида (например, несчастный пользователь сообщит вам: «Когда я ввел для вашей программы *эти* данные, она выдала мне *такие*

результаты...»), вы должны сказать себе: «Ну, наиболее очевидные места, где моя программа, судя по описанию проблемы, могла бы сделать что-то не так,— это подпрограммы А, В и С. Поэтому мне следует прежде всего проанализировать их работу».

Мы можем развить этот совет немного дальше. Если существуют несколько явных «подозреваемых», их следует проверять в порядке возрастания сложности. Так, в приведенном выше примере мы решили, что вероятные подозреваемые — это подпрограммы А, В и С; однако может оказаться, что подпрограмма А очень сложна, В средней сложности, а С относительно проста (либо потому, что содержит гораздо меньше операторов, либо потому, что выполняет более простую функцию, либо потому, что более тщательно документирована, либо просто потому, что вы лучше понимаете, как она работает). Таким образом, вы можете оценить, что вам потребуется пять минут, чтобы определить, есть ли ошибка в подпрограмме С, 30 мин,— есть ли она в подпрограмме В и три дня — есть ли она в подпрограмме А. Теперь должно быть достаточно ясно, какую подпрограмму следует анализировать первой.

8.1.7. Ничего не принимайте на веру

Выше мы говорили о том, что следует предполагать, что большинство ошибок в вашей программе — это результат ваших собственных оплошностей. Это, безусловно, верно, однако приходитвремя, когда вам нужно с пристрастием взглянуть на все остальное, что могло бы повлиятьна работу вашей программы. Речь идет о таких вещах, как аппаратные средства, оператор ЭВМ, операционная система фирмы-поставщика и даже погода (не одну программу во время ее выполнения испортили удары молний — не было ли это послание свыше?!).

Показательным примером ничем не оправданной веры может быть большая программа, состоящая из многих подпрограмм. Когда что-то идет неправильно, неискушенный программист вероятно воскликнет: «О, *эта* подпрограмма не может быть виноватой — она работает уже несколько месяцев!» И он одним махом отбросит половину подпрограмм, не считая их возможными источниками ошибки. Более опытный программист, с другой стороны, скажет, скорее всего: «Ну, я не думаю, что эта подпрограмма в чем-то виновата, но не проводились ли в ней за последнее время какие-либо изменения?» И почти наверное ответом будет: «Ну, было тут какое-то совсем пустяковое изменение — но это едва ли вызовет какие-либо проблемы, не так ли?» Затем опытный программист задаст следующий вопрос: обрабатывала ли ранее подозреваемая подпрограмма успешно такие входные данные, на которых выявилась программная ошибка? Подпрограммы, с успехом используемые в течение ряда лет, могут внезапно отказать, если подать новые входные данные (в качестве примера

приведем подпрограмму, печатающую даты юлианского календаря,— она может отказать 29 февраля високосного года, после того как с успехом использовалась в течение почти четырех лет).

В большинстве случаев ваши подозрения следует обращать в основнэм на программное обеспечение — все программные средства, которые дополняют вашу программуивзаимодействуют с ней. Если при выявлении какой-то ошибки в большой системе программирования обвиняют ваш модуль, то (предварительно убедившись в том, что в нем нет явной ошибки) вы должны проверить, не вносились ли какие-либэ изменения в соседние модули. Если вы проводили за последнее время изменения своего модуля, вам следует с особым пристрастием проверить также свое собственное произведение! Если и после этого вам не удастся найти ошибку, то вы и ваши коллеги должны спросить себя, были ли действительно хорошо проверены все остальные модули программы, кажущиеся ни в чем не повинными. Как и вдетективах, иногда виновницей ошибки окажется в конце концов самая незаметная и вполне благопристойная на вид подпрограмма.

Если вы все же не можете найти ошибку таким образом, вам следует начать с пристрастием искать других потенциальных виновников, от которых также зависит работоспособность программных средств: это аппаратные средства ЭВМ, оператор, телефонная компания, базы данных или даже ваш собственный шеф! Никогда не забывайте, однако, что ошибка в конце концов может оказаться в вашем собственном модуле; поэтому, хотя вы должны иногда подозревать других, делайте это вежливо и корректно — и будьте смиренны и полны раскаяния, еслиошибкаобнаружитсяподвашим собственным носом.

8.1.8. Используйте „маленькие клеткисерого вещества"

Герой многих детективов Агаты Кристи, легендарный Эркюль Пуаро, неоднократно замечал своим друзьям, что истинно сложные и запутанные преступления раскрываются не ищейками и облавами. В то время как полиция деятельно анализирует следы преступления, выслеживает подозреваемых, опрашивает свидетелей и обычно ходит вокруг да около в поисках убийцы, Пуаро сидит в своем удобном кресле и *размышляет*. Прямолинейные, трудоемкие и методичные способы, применяемые полицией, играют, конечно, важную роль, считает он, но его лично больше интересуют психологичесике мотивы преступления, психология жертвы и в конце концов убийцы.

Аналогия следствия с отладкой достаточно ясна. Дампинги памяти, трассировочные таблицы и контрольные выводы («фотографии»), а также та таинственная процедура, которая известна под названием «бумажная отладка»,— все это играет свою роль, не не является самоцелью. Это только средства для достижения цели, которая состоит в отыскании ошибки в программе. Довольно жалкий вид име-

ют некоторые молодые программисты, восклицающие, например: «Я получил дамп памяти, что же делать теперь?» Как они разочарованы, что ошибка не «выскочила» из массы шестнадцатеричных цифр, выведенных на печатающее устройство, и не бросилась им прямо в глаза!

Если развить аналогию с детективом чуть дальше, то часто имеет смысл отказаться от дампов памяти, трассировочных таблиц и контрольных выводов. При поиске сложных ошибок можно достичь гораздо лучших результатов, если положить ноги на стол, уставиться в потолок и поразмыслить о природе ошибки. «Большинство ошибок имеет свою «психологию», т. е. они, скорее всего, характеризуются, конкретным *«поведением»*. Быть может, вы говорите себе, после того как какая-то ошибка второй или третий раз задрала свою гадкую маленькую головку: «Смотри, опять *эта* ошибка! Она, кажется, всегда проявляется сразу же после того, как мы открываем второй входной файл». Теперь самое время сесть и спокойно подумать: Чем существенным характеризуется открытие второго входного файла? Что происходит между открытиями первого и второго входного файлов? Встречались ли вам какие-либо другие ошибки, связанные с открытием файла? (Еще одна героиня детективов Агаты Кристи — мисс Джейн Марпл — всегда старалась найти у подозреваемых в убийстве черты, роднящие их с людьми, которых она знала в небольшой английской деревне, где жила.) Какое следующее действие должна была бы выполнить программа? Есть ли какие-либо другие подозрительные обстоятельства, окружающие «безвременную кончину» программы?

8.1.9. Беседуйте со своими друзьями

Если у вас встречаются затруднения при поиске ошибки в программе, иногда бывает полезно поговорить об этом с приятелем. Если некоторые из ваших коллег работают над тем же самым проектом, они, скорее всего, должны иметь общее представление о вашей проблеме. Фактически, именно в этом состоит одна из причин, почему многие руководители проектов и заведующие отделами программирования пытаются сделать так, чтобы каждый модуль знали и понимали несколько программистов (это помогает также выйти из положения, в случае, когда основной автор — разработчик модуля — переходит на другую работу).

Если даже никто непосредственно в вашей программе раньше и не разбирался, для вас все же будет полезно, если кто-то просмотрит вашу рабочую программу. Бывает так, что вы сужаете область поиска до конкретной подпрограммы, но потом оказываетесь не в состоянии обнаружить ничего плохого в своей программе. Чем больше вы смотрите на листинг программы, тем больше убеждаетесь, что ничего плохого здесь нет. В этот момент бывает полезным, если вашу

программу просмотрит кто-то еще, даже если он с ней совершенно не знаком. Опишите программу, характер проблемы и последовательность шагов, через которую проходит программа. Часто бывает, что уже только такое объяснение работы программы приводит к обнаружению ошибки; в других случаях этому могут помочь несколько умных вопросов вашего коллеги (например, «Вы уверены, что задали нужное исходное значение этой переменной перед началом вычислений?»). Еще в некоторых случаях ваш коллега может просто взглянуть на листинг программы и указать ошибку, на которую вы многократно смотрели, но не видели.

Здесь уместно сделать следующее предупреждение: далеко не каждый является хорошим «партнером по отладке»; фактически, некоторые могут оказаться хуже, чем бесполезными. Некоторые из ваших коллег будут вежливо выслушивать описание вашей проблемы, но будут не в состоянии помочь какими-либо полезными советами. Другие не будут даже делать вид, что вежливо слушают. Вы обнаружите, что некоторые из коллег имеют стиль программирования и отладки, столь разительно отличающийся от вашего, что их помощь вам только повредит. Другие не смогут быстро понять природу проблемы и степень вашего продвижения в процессе отыскания ошибки; они будут только раздражать вас своими безнадежно наивными вопросами, хотя и заданными с самыми благими намерениями. Наконец, даже самым отзывчивым и обязательным из ваших коллег надоест смотреть вашу программу, если вы будете кричать «караул» слишком часто.

Мораль, по-видимому, ясна; стоит потратить какое-то время, чтобы определить, кто из ваших коллег готов и способен помогать вам в поиске ошибок. Когда вы это определите, обращайтесь к ним только в крайних случаях и всегда показывайте, как высоко вы цените их помощь. И по меньшей мере вам следует отвечать любезностью на любезность, когда они будут искать ошибки в своих программах.

8.1.10. Проверяйте свои подозрения с помощью тест-программ

В различные моменты в процессе отладки у вас будут появляться сомнения относительно того, действительно ли аппаратные средства, набор команд машинного языка, язык программирования или операционная система ведут себя в определенных аспектах так, как вы это предполагаете. Так, вы можете спросить себя: «Что *реально* происходит, когда у меня получается ошибка четности при команде ОБРАТНОЕ ЧТЕНИЕ для магнитной ленты? Быть может, операционная система автоматически пропускает плохую запись, прежде чем выдать ответ с признаком ошибки моей программы? Если это было бы так, то легко объяснялось, почему у меня столько затруднений».

В подобной ситуации вы, скорее всего, обнаружите, что документация, касающаяся вашего вопроса, написана неясно, неточна, вводит в заблуждение, явно неправильна или попросту отсутствует. Если даже в документации все правильно, вы, быть может, неверно поняли то, что в ней написано. Поэтому часто имеет смысл написать небольшую тест-программу, чтобы проверить свои подозрения. Такую небольшую и простую тест-программу обычно можно написать за несколько минут; она требует очень малых затрат машинного времени и обычно не вызывает собственных проблем тестирования. Если предположение, которое вы собираетесь проверить, слишком сложно, то, быть может, будет легче изменить прохождение отлаживаемой вами исходной программы; для решения проблемы может оказаться вполне достаточным дамп оперативной памяти в соответствующей точке вашей программы.

Важно помнить, что здесь вы применяете следующий методологический принцип: вместо того чтобы пытаться найти некоторую ошибку, *вы допускаете, что не знаете, как выполняется конкретная аппаратная функция (или функция операционной системы и т. д.),* поэтому вы проводите эксперимент. Независимо от того, положителен или отрицателен результат эксперимента, он должен дать вам дополнительную информацию, а в большинстве случаев чем больше у вас информации, тем легче становится процесс отладки.

8.1.11. Отключайтесь и откладывайте проблему до завтра

Бывают случаи, когда ни одна из рекомендаций этой главы не приносит никакой пользы. После того как вы проработали с листингом программы много часов, выкурили много пачек сигарет или выпили несколько галлонов кофе, мозг просто отключается, если даже вы не хотите себе в этом признаться. В этом момент (который может наступить в три часа ночи или в три часа дня) самое лучшее — это бросить все, совершенно забыть о программе и провести несколько часов, делая что-нибудь, что позволит вам расслабиться: поспать, поиграть в гольф, понаблюдать за птичками, выпить пива или отвлечься каким-либо другим, иногда весьма странным способом, каких много в запасе у. программистов. Если ваш начальник имеет хотя бы минимальный опыт программирования, он все поймет и одобрит; если нет, не обращайте на него внимания или предложите ему самому найти эту программную ошибку[1)].

[1)] В равной степени хорошо при этом начать вести себя с некоторым налетом безумия, поскольку ни один руководитель не хочет, чтобы его обвинили в том, что он доводит своих программистов до полного сумасшествия, хотя он знает, что большинство из них несколько неврастенично даже в свои лучшие времена. В один из особенно острых авралов при отладке одной моей системы я как-то появился на работе только в три часа дня; принеся с собой банку краски и кисть, я выкрасил

Такой подход часто дает неожиданную выгоду: пока вы отдыхаете, ваш мозг часто подсознательно продолжает обдумывать проблему; действительно, когда вы ложитесь спать после трудного дня и вечера отладки, ваша программа может вам даже присниться. Когда вы вернетесь к работе на следующий день, ошибка зачастую уже смотрит на вас из листинга программы и ждет, чтобы вы ее поймали; если нет, то вы по крайней мере хорошо выспитесь за ночь.

8.2. Типичные ошибки и погрешности программирования

Можно, по-видимому, с уверенностью сказать, что существует бесконечно много погрешностей и ошибок, которые совершают программисты, точно так же, как существует громадное число способов решения или написания программы для решения конкретной задачи. Тем не менее нередко бывает, что каждый программист попадает в одни и те же ловушки, допускает одни и те же оплошности и встречается с одними и теми же проблемами.

Для любой организации было бы весьма полезно собирать статистические данные по ошибкам и разбивать их на категории. В результате можно было бы составить перечень наиболее часто совершаемых ошибок, чтобы каждый программист мог ознакомиться с этим перечнем, прежде чем обращаться за помощью. Здесь нам требуются более достоверные статистические данные; кто-то должен, наверное, проводить их анализ, чтобы определить процентные соотношения ошибок, вызываемых различными причинами. Не совсем ясно, правда, что программист должен делать с подобным перечнем. Если он знает, что его программа работает неправильно (например, когда она выбрасывается из машины в середине прогона), то, как он может сказать, какая из «десяти наиболее распространенных ошибок» имеет место? Аналогично, как можно научить начинающего программиста сопоставлять неправильное поведение своей программы с этими типичными ошибками?

В этой области — большой простор для возможных исследований и последующего анализа. Однако в качестве отправной точки полезным может оказаться следующий перечень:

1. *Ошибки перфорирования и переписки.* Значительное число ошибок — это простые ошибки перфорирования; например, часто путают цифру нуль и букву О, меняют местами символы, по той или иной причине пропускают операторы и т. д.

2. *Переменные величины без начальных значений.* Еще одна из наиболее типичных ошибок касается переменных величин, которые

дверь своей комнаты в ярко-зеленый цвет, а затем ушел домой на весь день. Никто не сказал ни слова, за исключением смотрителя здания, который забеспокоился, как бы кто-нибудь из других программистов не захотел повторить это со своими дверьми.

используются до того, как проведена их.инициализация. Эту проблему, кажется, можно было бы решить с помощью более совершенной аппаратной логики, например, заставив транслятор или операционную систему устанавливать для всех программных переменных некоторое начальное значение, вызывающее прерывание, если сама программа не произведет нужную повторную инициализацию.

3. *Использование переменных многими модулями.* Об этом упоминалось в одной из предыдущих глав, и это является, по мнению автора, одной из самых распространенных программных ошибок. Ошибка происходит тогда, когда модуль А и модуль В пытаются использовать одну и ту же переменную или ячейку памяти в качестве рабочего регистра для хранения промежуточных результатов. Почти неизбежно в какой-то ситуации один из модулей испортит то, что другой записал в эту временную память. Проблема значительно усложняется в системах реального времени, где модуль А и модуль В могут прерывать друг друга в дополнение к вызову друг друга (или вместо вызова).

4. *Ошибки управления и логики.* Я не знаю, как назвать их более точно, но, по-видимому, это неплохое название еще одного распространенноготипаошибок. Программист пытается вызвать модули, вызов которых не был предусмотрен, или не вызывает модули, когда это нужно было бы сделать; он выбирает неверный путь при условном переходе и т. д. и т. п.

5. *Индексация с выходом за границы массива.* Это очевидная, но довольно часто встречающаяся ошибка: программист начинает обращаться к ячейкам, расположенным за границами массива, что приводит к различным последствиям. Иногда его программа выбрасывается (если он пытается обращаться к «чужим» ячейкам вне выделенной ему области памяти) или он попадает на данные другого массива и т. д. Это еще одна проблема, которую следовало бы решать аппаратным способом, как делается в машинах фирмы Burroughs. Помочь избежать подобных проблем могут также некоторые специфические дополнительные средства контроля ошибок, имеющиеся в машинных языках и трансляторах.

6. *Неправильное окончание программных последовательностей.* Быть может, одним из лучших примеров этого является процедура слияния N массивов. У программиста часто встречаются затруднения, когда читаются последние несколько записей, некоторые из файлов уже исчерпаны (появились признаки «конец файла») и т. д.

7. *Не предусмотрены особые ситуации.* Часто оказывается, что программист составляет свою программу только для очевидного случая (быть может, случая, указанного в техническом задании), не предусматривая возможности исключительных или ошибочных ситуаций. В таких случаях поведение его программы обычно непредсказуемо.

8. *Неверные связи и интерфейсы.* Это еще одна типичная проблема,

причем проблема, решение которой является основной целью нисходящего тестирования: от одного модуля к другому передается неправильная информация, модули «забывают» засылать в память и восстанавливать содержимое своих регистров и т. д.

9. *Неверные форматы данных.* Программисту, особенно работающему на таких языках, как Кобол или ПЛ/1, очень легко сделать ошибки при выборе масштаба (шкалы) типа или формата числовых переменных своей программы.

10. *Обращение с константами, как с параметрами.* Во многих высокоуровневых языках программирования операторы вида

```
CALL GLOP(A, B, 3)
```

могут доставить много хлопот. Если подпрограмма GLOP считает свой третий аргумент переменной величиной, она может попытаться изменить ее значение, тем самым, скорее всего, испортив «литерал» 3.

11. *Неправильное употребление сложных булевых выражений.* Выше мы уже говорили о том, как легко могут ошибаться программисты при использовании операторов вида

```
IF A AND NOT EQUAL C OR D
```

Другие выражения, где встречаются вложенные операторы AND (И) и OR (ИЛИ), могут приводить к ошибкам, если они не разделяются скобками на соответствующие части.

12. *Неправильное употребление вложенных операторов IF.* Вложенные операторы ЕСЛИ обычно вызывают так много ошибок, что большое число организаций, работающих на Коболе, запрещает их использование. Программисты, работающие на Алголе и на ПЛ/1, по-видимому, несколько привычнее к подобным операторам, но они также время от времени испытывают затруднения с их применением.

13. *Некорректный выход из подпрограмм.* Эта проблема свойственна главным образом языку ассемблера и Коболу, например, при использовании перекрывающихся операторов PERFORM; в меньшей степени она свойственна тем языкам, где структура подпрограмм более четко формализована.

14. *Обращение к областям данных после операторов WRITE.* Одна из типичных проблем здесь заключается в том, что области данных или буферные области оперативной памяти освобождаются операционной системой (или иногда очищаются) после копирования с помощью оператора WRITE. Любая попытка программиста получить значащие данные при обращении к этим областям может поэтому вызвать большую путаницу.

15. *Проблемы, связанные с флажками.* Об этой проблеме шла речь в гл. 6: если программист использует большое количество флажков (признаков), то, по всей вероятности, будет существовать несколько недопустимых их сочетаний, так что неправильная установка ка-

кого-либо флажка приведет к тому, что программа в течениедовольно длительного времени будет выполняться для бессмысленного «состояния», прежде чем, наконец, зто выявится.

16. *Непредусмотренные особые случаи ввода-вывода.* Наиболее распространенная ситуация здесь — это отсутствие средств, предусматривающих соответствующую обработку сигналов конца файла и ошибки четности при операциях ввода-вывода.

17. *Отсутствие анализа кодов ответа.* Операционная система фирмы-поставщика реализует большинство операций ввода-вывода и ряд других важных служебных функций. Когда управление возвращается прикладной программе, некий «код ответа» обычно показывает, успешно ли была выполнена соответствующая операция. Многие программисты забывают проанализировать этот код или предполагают, что никогда не бывает ошибок ответа; поэтому в случае, когда ошибка ответа все же *происходит,* их программа продолжает работать так, как будто бы ничего не случилось.

18. *Счетчики слишком малой разрядности.* Типичная проблема для многих экономических приложений с программированием на языке Кобол. Если программист выделил для счетчика ограниченное место, то легко может произойти переполнение (возможность, анализ которой зачастую не предусматривается) и сброс счетчика в нуль.

19. *Проблемы адресации.* При работе на языке ассемблера программист часто ошибается, используя косвенную адресацию, модификацию и относительную адресацию, предусмотренные только в более сложных машинах.

20. *Ошибки в сложных выражениях.* Эта проблема упоминалась в гл. 5; она, по-видимому, свойственна в основном научно-техническим приложениям. Даже при щедром использовании скобок программисты часто получают в результате вычислений далеко не то, что хотели.

21. *Проблемывыравнивания.* Этотипичная проблема языка высокого уровня, особенно Кобола. При пересылке последовательностей символов, сравнений последовательностей символов или выполнении определенных типов арифметических операций выравнивание переменных может играть большую роль. Для машин, работающих со словами фиксированной длины, подобные проблемы часто преувеличивают.

22. *Выход за верхнюю или нижнюю границы программного стека.* Первый случай возникает, если у программиста получился более глубокий уровень вложенности или рекурсии, чем он планировал первоначально (или более глубокий, чем допускает система); во втором случае обнаруживается определенная ошибка — делается попытка выхода из подпрограммы, которая не была вызвана.

23. *Ошибочное использование общих регистров для базирования.* По очевидным причинам эта проблема, доставляющая больше всего

хлопот программистам, работающим на языке ассемблера на таких машинах, как машины Систем IBM/360 и IBM/370. Это может быть также проблемой для других машин с многочисленными общими регистрами и регистрами смещения.

24. *Попытка работать с данными как с программой.* Это происходит обычно в случае, когда программа делает «сумасбродный» переход в область данных или когда последовательность закодированных команд переходит в какие-то данные. Это может также быть следствием индексации с выходом за границы массива, о чем говорилось выше. Если программа записала данные после конца массива, эти данные могут попасть в область, ранее занятую верными машинными командами.

25. *Использование неправильной версии программы.* Рано или поздно это обязательно случается с каждым. Ситуация становится еще более запутанной, если программист работает с листингом программы, объектным модулем и исходной программой, которые не соответствуют друг другу.

26. *Отсутствие ограничителя (разделителя).* Во многих языках программирования в качестве признака конца комментария или оператора программы используется символ «точка с запятой»(;). В других случаях ограничителем оператора, группы операторов, макроопределения или даже целой программы является специальное слово END (КОНЕЦ). Если программист забудет вставить в нужное место точку с запятой или оператор КОНЕЦ, это может вызвать большие неприятности, например компилятор может воспринять группу программных операторов как часть длинного комментария.

8.3. Классические способы и приемы отладки

Обсудив некоторые общие *принципы* отладки, обратимся теперь к некоторым *способам и приемам* отладки. Хотя многие из этих, способов могут быть вам уже знакомы, есть ряд интересных вариантов трассировок и контрольных выводов, которым стоит уделить внимание. Здесь будут в основном рассматриваться не детали самих пакетов отладочных средств, поскольку эту информацию легко можно найти в поставляемых фирмами руководствах по программированию. Вместо этого основное внимание будет уделено программным ситуациям, в которых эти пакеты могли бы оказаться полезными.

8.3.1. Дампинги памяти

Одним из самых древних и самых полезных приемов отладки является простой дампинг (осуществление распечатки) оперативной памяти. В любой момент в процессе выполнения программы программист может, если он считает это полезным, распечатать содержимое оперативной памяти (или выбранного участка памяти) на быстро-

действующем печатающем устройстве. Возможен и вариант, когда содержимое памяти записывают на магнитную ленту, диск или барабан, чтобы распечатать впоследствии. Анализируя саму распечатку (дамп) памяти, программист обычно может очень хорошо понять природу ошибки в своей программе.

Полезность распечатки памяти нельзя отрицать, однако иногда наблюдается тенденция к удивительному злоупотреблению этой возможностью. Если природа программной ошибки не ясна сразу же, программист часто распечатывает *полностью* содержимое памяти, а затем удаляется на свое рабочее место с листингом 5-сантиметровой толщины, ожидая вдохновения свыше. Если оно не приходит достаточно скоро, то не исключено, что он мчится в машинный зал ЭВМ и снова производит распечатку памяти (ведь первый дамп оперативной памяти, бытьможет, неверный!?!). Если у вас наблюдаются такие же стремления к распечатке памяти, то имейте в виду, что дажевлучшемслучае90% или более отпечатанной информации будет бесполезной и к проблеме не относящейся; в худшем случае весь дамп памяти для решения проблемы ничего не даст. Распечатки памяти очень похожи на полицейские облавы: если они производятся в нужный момент, то быть может, удастся захватить одного-двоихподозрительных; еслиже моментвыбраннеправильно, то никакого полезного результата не будет. Имейте также в виду, что распечатки оперативной памяти — не бесплатное удовольствие: они отнимают известное время ЦП (правда, в большинстве случаев довольно небольшое) и потенциально большое время ввода-вывода (например, время каналов ввода-вывода, быстродействующего печатающего устройства и (или) ленты и т. д.).

Если вы решаете распечатать содержимое оперативной памяти, удостоверьтесь вначале, что дамп имеет удобочитаемый формат: как бы ни был умен человек, ему довольно сложно и утомительно разбираться в шестнадцатеричном дампе (как заметил один программист, если бы подразумевалось, что мы должны мыслить в шестнадцатеричном коде, то природа дала бы нам шестнадцать пальцев). Примерами средств для печати в формате могут служить подсистемы DUMP и PDUMP в Фортране. Они дают программисту возможность распечатывать содержимое выбранных участков памяти в формате целых величин, чисел с плавающей точкой, в восьмеричном представлении или в алфавитно-цифровом виде. Если ваша машина или ваш язык программирования таких возможностей не предоставляют, вам следует подумать о разработке универсальной подпрограммы распечатки, которой могли бы пользоваться также ваши коллеги (вспомните о том, что обсуждалось в гл. 3).

Еще одна рекомендация относительно дампов памяти: сохраняйте их (либо в обычном печатном виде, либо на магнитной ленте) для будущих справок. Если вам встретится исключительно тонкая ошибка, ее поиск может занять несколько недель или месяцев;

вы можете захотеть вернуться к предыдущему дампу, чтобы посмотреть, проявляется ли эта ошибка какими-либо своими характерными приметами. При этом естественно предполагать также, что вы должны вести соответствующие записи, показывающие, когда получен дамп, при каких обстоятельствах, в каком файле на магнитной ленте или на диске он хранится и т. д.

Наконец, подчеркнем мысль, высказанную еще в гл. 1: если вы хотите, чтобы распечатка оперативной памяти дала вам полезную информацию, старайтесь избегать совместного использования одних и тех же рабочих областей памяти несколькими подпрограммами. Если одна и та же рабочая область памяти используется совместно несколькими подпрограммами, то дамп памяти покажет только последнюю записанную информацию, а следы предыдущих действий других подпрограмм будут утеряны.

8.3.2. Трассировки

Другой весьма распространенный прием отладки — это так называемая трассировка, известная также под наименованием «моментальная фотография». Большинство изготовителей ЭВМ поставляют трассировочные средства для различных высокоуровневых и низкоуровневых языков программирования. В действительности трассировка — это просто разновидность дампа оперативной памяти, описанного в предыдущем подразделе. Трассировочные средства позволяют распечатывать содержимое определенных ячеек памяти (переменных программы, параметров связи подпрограмм и т. д.) в моменты или при условиях, указанных программистом. Эта информация обычно выводится на быстродействующее печатающее устройство или записывается на магнитную ленту вывода, чтобы программист мог ее впоследствии проанализировать.

Практически все соображения, высказанные относительно дампинга памяти, приложимы и к трассировке. При трассировке, если ее применять без разбора, расходуются стопы бумаги (во многих случаях даже больше, чем придампингах, поскольку трассировка представляет динамический процесс прослеживания хода программы в отличие от однократного, статического дампа памяти), причем получается мало полезной информации. Трассировка занимает также, скорее всего, значительное количество времени ЦП (поскольку ошибочная программа идет и идет без конца) и ресурсов ввода-вывода. Как и в случае дампов памяти, некоторые трассировки представляются на печати в шестнадцатеричном коде, поэтому их почти невозможно расшифровывать (если, конечно, не иметь шестнадцати пальцев...).

Сам принцип трассировки — слишком общий, однако существует несколько более или менее «стандартных» видов трассировок. Все они довольно просты для реализации и полезны в большин-

стве наиболее распространенных ситуаций программирования. В качестве исходного пункта для создания библиотеки отладочных программных пакетов рекомендуется следующий, хотя и не исчерпывающий перечень трассировочных средств:

1. *Трассировка для подпрограммы.* Назначение этой трассировки — отпечатать индикационные признаки каждой ситуации, в которой осуществлялся вызов конкретной подпрограммы. При такой трассировке можно было бы, по-видимому, печатать все аргументы, представляемые подпрограмме, и даже признаки того, осуществила ли подпрограмма нормальный выход на вызывающую программу или нет.

2. *Трассировка для переменной.* Этатрассировка применяется для распечатки значений указанных переменных, элементов данных, областей рабочей памяти, массивов и т. д. В большинстве случаев достаточно печатать значения переменной только при изменениях; однако могут быть ситуации, когда программисту хотелось бы проследить *все* обращения к переменной.

3. *Трассировка для оператора.* Такую трассировку нужно считать вынужденным последним средством, которое применяется только тогда, когда все другие виды отладочных приемов не позволили локализовать ошибку. Здесь печатаются значения всех переменных (и регистров, сумматоров и т. д.), которые затрагиваются некоторым программным оператором как до, так и после выполнения этого оператора.

4. *Трассировка для макрокоманды.* Для программистов, испытывающих затруднения в сопряжении своих прикладных программ с операционной системой, такая трассировка покажет содержимое соответствующих регистров до и после «макрокоманды», или обращения к операционной системе. Такая трассировка могла бы также показать, произошел или нет ошибочный возврат от операционной системы к прикладной программе. (Отсутствие контроля ошибочных кодов ответа — распространенная ошибка в этой области.)

5. *Трассировка для терминала.* Для систем реального времени, коммутации сообщений, разделения времени и других типов систем, работающих с терминалами, при отладке часто требуется точно знать, какие входные данные поступают от конкретного терминала (или группы терминалов). Такая трассировка показывает все входные и выходные сообщения для конкретного терминала или, быть может, группы терминалов. Поскольку многие подобные системы реального времени уже предусматривают запись *всех* сообщений терминалов на магнитную ленту или диск с целью последующего восстановления хода событий, такую трассировку можно было бы просто реализовать одной программой, выбирающей нужные сообщения с ленты, где хранится оперативная протокольная информация, и распечатывающей их на быстродействующем печатающем устройстве.

6. *Трассировка для базы данных.* Такая трассировка была бы полезной для систем, включающих крупные и сложные базы данных, особенно для систем с центральным «администратором базы данных», который координирует всю работу базы данных, все операции по созданию новых файлов, по уничтожению записей, *все* обращения к базе данных или любые другие допустимые сочетания операций.

7. *Трассировка по времени.* В некоторых программных системах реального времени иногда трудно даже понять, что нужно прослеживать, поскольку ошибка может оказаться столь тонкой, что ее нельзя легко «привязать» к конкретной переменной или подпрограмме. Для этого случая, по-видимому, желательно иметь трассировочную программу, периодически печатающую (или записывающую на диск, чтобы ускорить процедуру) содержимое выбранных участков памяти. Если ошибку просто невозможно найти, иногда может помочь даже проведение трассировок по *случайному* временному закону.

8. *Условные трассировки.* В некоторых случаях программист может знать, что ему нужна только та информация, которая соответствует выполнению определенных условий, например, когда переменная А имеет положительное значение, а переменная В — отрицательное. Такие условия, если они не слишком сложны, можно было бы задать для *условной трассировки,* которая могла бы иметь также и свойства многих других типов трассировок, упомянутых выше.

Основная проблема, связаннаястрассировочнымиоператорами, для большинства высокоуровневых языков программирования состоит в том, что они обычно вставляются в программу во время компиляции. Таким образом, если программист делает трассировку для одной части программы и выясняет, что он не выявил какую-то ошибку, он должен вставить трассировочный оператор в некоторое другое место, *произвести перекомпиляцию* и попробовать снова. На это будет, безусловно, затрачено значительное машинное время. Во многих случаях лучший способ заключается в том, чтобы вставить *несколько* трассировочных операторов в программу, а затем с помощью операторов ЕСЛИ решать, для каких переменных следует выполнить трассировки. При этом программист мог бы в начале выполнения программы прочитать входную перфокарту и на основании содержащихся в ней данных определить, для какой части программы нужна трассировка.

8.4. ДДТ-подсистемы динамической отладки

Обсудив некоторые из более традиционных средств и приемов отладки, обратимся теперь к рассмотрению подсистем отладки, которые до сих пор имели в области вычислительных машин несколько ограниченное применение, но которые, по мнению многих специа-

листов (включая и автора), являются гораздо более мощным и эффективным инструментом, чем дампы оперативной памяти и трассировки. Первоначальное их название — «средства динамической отладки», но известны они в основном по своим популярным инициалам — ДДТ (Dynamic Debugging Technique). Это название явно и намеренно выбрано по аналогии с хорошо известным средством против насекомых: подсистема ДДТ помогает столь же эффективно избавляться от программных ошибок (англ. bugs), как порошок ДДТ — от насекомых (англ. также bugs).

В этом разделе мы опишем некоторые из разнообразных видов подсистем или пакетов ДДТ, которые могут быть реализованы. Мы опишем также некоторые свойства, общие почти для всех версий ДДТ. Наконец, мы более детально расскажем о реализации одной простой версии ДДТ.

8.4.1. Типы пакетов ДДТ

Существуют три основных типа пакетов ДДТ, которые можно различить по способу обработки входных сообщений, поступающих с терминала:

1. *Автономный пакет ДДТ.* Это простейший вид ДДТ и его часто применяют для отладки программ на небольших мини-машинах. Как показывает его название, автономный пакет ДДТ не зависит от операционной системы и не работает одновременно с какими-либо другими программами; он действительно самостоятелен.

Отличительной характеристикой автономного ДДТ является то, что он сам обеспечивает весь диалог с программистом, работающим на терминале. Другими словами, он принимает с терминала входное сообщение от программиста без помощи операционной системы и располагает всеми возможностями для передачи выходного сообщения программисту. Более того, он не позволяет производить каких-либо вычислений, пока сам ожидает входного сообщения от программиста. Это очень важный момент: здесь *машина находится в холостом режиме, ожидая входного сообщения от программиста*. Такое решение приводит к довольно неэффективному использованию имеющегося машинного времени, поскольку программист, скорее всего, подолгу смотрит в потолок, раздумывая, что же делать дальше. Однако подобное практическое решение часто приемлемо при отладке программ на мини-компьютере, где нет других программистов, ожидающих реакции машины на свои действия. Иногда этот режим полезен также при отладке программ реального времени, поскольку единственный рациональный вариант здесь состоит в том, чтобы использовать для отладки пульт машины.

Важно подчеркнуть, однако, что не сам по себе автономный пакет ДДТ виноват в неэффективном использовании машинного времени ЦП, а скорее то, что он не позволяет программисту ана-

лизировать и модифицировать свою программу в процессе ее реального выполнения. В любой конкретный момент работает либо ДДТ, либо отлаживаемая программа, одновременное их выполнение невозможно.

2. ДДТ с разделением времени. Этот второй вид ДДТ отличается от автономного ДДТ только тем, что принимает сообщения с терминала через операционную систему и работает с отлаживаемой программой, находясь под управлением операционной системы.

Это означает, что операционная система может выполнять другие действия, ожидая, пока программист выдаст с терминала входное сообщение для пакета ДДТ с разделением времени. Такой режим весьма распространен в системах с разделением времени, где несколько программистов могут одновременно отлаживать свои программы, работая с собственной копией ДДТ каждый (или, быть может, с одной реентерабельной копией совместно).

Дополнительное достоинство версии ДДТ с разделением времени заключается в том, что она работает под защитой операционной системы. Поэтому, если программа пользователя начинает «дурить» (как время от времени бывает со всеми программами), операционная система предотвращает выполнение этой программой недопустимых команд или ее обращение к незаконным участкам памяти. Отметим также, что если говорить о программисте-пользователе, то пакет ДДТ *не может* работать одновременно с его программой; здесь либо идет его программа, либо — программа ДДТ с разделением времени, но не обе сразу. Мультипрограммные характеристики, свойственные обычно операционной системе, могут использоваться только *другими* пользователями, заданиями или прикладными программами.

3. *ДДТ реального времени.* Наиболее сложный вид ДДТ —это пакет, который позволяет отлаживаемым программам выполняться в то время, когда сам пакет ожидает входное сообщение с терминала программиста. Хотя подобный пакет ДДТ можно создать для прикладных программистов, используя многопрограммные режимы, предусмотренные аппаратными средствами некоторых фирм-поставщиков, это делается крайне редко. Версии ДДТ реального времени вместоэтогоразрабатываютсяобычнодляупрощения отладки мониторов, операционных систем и прочих систем программирования реального времени. Как мы уже указывали, различные виды ДДТ характеризуются различными способами обмена сообщениями с терминалом. В случае ДДТ реального времени в качестве терминала служит обычно *пультовый телетайп,* который располагается вблизи машинного пульта и непосредственно подключается в ЭВМ; ДДТ реального времени играет роль программы управления вводом-выводом для пультового телетайпа.

Таким образом, пока программист сидит в раздумье за пультом, вся остальная система может свободно и беспрепятственно работать.

Когда программист печатает на телетайпе какое-то сообщение, вырабатывается сигнал прерывания и управление процессором передается пакету ДДТ реального времени с помощью той или иной схемы приоритетного прерывания. Послеобработкизапроса программиста (на просмотр содержимого памяти, изменение содержимого памяти и т. д.) ДДТ освобождает аппаратуру прерывания, и остальная система продолжает работать.

Ниже мы опишем универсальные черты пакетов ДДТ; при необходимости будут отмечаться специфические особенности этих черт для автономного ДДТ, ДДТ с разделением времени и ДДТ реального времени.

8.4.2. Общие черты пакетов ДДТ

Предположим для целей нашего обсуждения, что программа, отлаживаемая с помощью ДДТ (и называемая часто «проблемной программой»), написана *на языке ассемблера,* поскольку большинство имеющихся в настоящеее время версий ДДТ позволяют вести отладку только на языке ассемблера. Можно создать подобные ДДТ средства отладки и для таких языков высокого уровня, как Фортран, Кобол и ПЛ/1 (недавно были разработаны несколько подсистем отладки для вычислительных центров, обслуживающих абонентов с разделением времени). Однако эти высокоуровневые версии ДДТ обычно требуют, чтобы проблемная программа не компилировалась, а *интерпретировалась.*

Обычно ДДТ загружаетсяв машину вместе с подлежащей отладке программой (программами). При этом получается распределение памяти, показанное на рис. 8.4. Пакет ДДТ ведет диалог с программистом, сидящим за своим терминалом, в одном или нескольких стандартных *режимах.* Более простые версии ДДТ могут предусматривать толко один разрешенный режим, например восьмеричный. Как правило, однако, программисту представляется возможность выбрать один из нескольких различных режимов отладки:

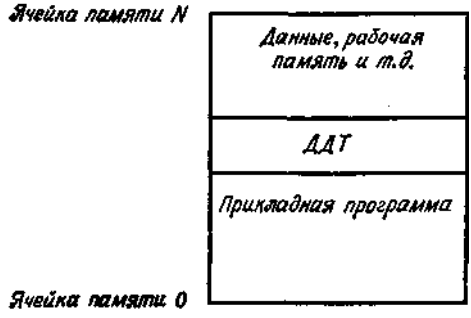


Рис. 8.4. Типичное распределение оперативной памяти при работе с ДДТ.

1. Режим отладки на языке ассемблера (символическом).
2. Числовой режим с представлениемчисел в одном из указанных форматов (восьмеричном, десятичном, шестнадцатеричном или с плавающей точкой).
3. Текстовый режим (в коде ASCII, EBCDIC и т. д.). Нужный режим задается пакету ДДТ специальной директивой; при отсутствии специальных указаний ДДТ работает в своем стандартном, режиме (обычно восьмеричном или символическом).

В распоряжении программиста имеется несколько типов *директив,* с помощью которых он может давать указания пакету ДДТ, чтобу получить нужную помощь в отладке программы. К наиболее употребительным директивам, которые будут описаны ниже, относятся:

1. Директивы просмотра и изменения содержимого ячеек памяти.
2. Директивы включения и исключения контрольных остановов.
3. Директива «GO» («пуск» проблемной программы).
4. Директивы поиска.

8.4.3. Директивы ДДТ для просмотра содержимого памяти

Даже самые примитивные программы ДДТ обладают способностью опросить и изменить содержимое любой ячейки памяти, доступной программисту. Адрес ячейки памяти, содержимое которой хочет узнать программист, в некоторых версиях ДДТ задается как восьмеричный параметр, в то время как в других используются символические таблицы, формируемые ассемблером, чтобы программист мог указывать ячейку памяти с помощью символического выражения.

Почти во всех версиях ДДТ в качестве директивы опроса ячейки памяти используется *символ косой черты* (т. е. символ «/»). Так, чтобы опросить ячейку памяти 234, программист набирает

```
234/
```

на своем терминале. При работе с версией ДДТ, рассчитанной на символические входные сообщения, он мог бы напечатать

```
XYZ + 3/
```

чтобы опросить третью ячейку после ячейки, символически обозначенной в его программе меткой «XYZ».

Получая директиву опроса ячейки памяти, ДДТ реагирует на нее, печатая текущее содержимое этой ячейки. Таким образом, когда программист просит посмотреть ячейку 234, происходит следующий диалог:

```
234/  1234
```

где выходное сообщение ДДТ подчеркнуто, чтобы его можно было отличить от входного сообщения программиста. При более совершенной версии ДДТ диалог мог бы происходить следующим образом:

XYZ + 3/ ADD ABC+1

В обоих случаях программист запрашивает информацию одного и того же типа; во втором случае, однако, ДДТ печатает содержимое ячейки памяти в символическом представлении, в формате языка ассемблера.

Отметим, что на две последовательные директивы опроса одной и той же ячейки памяти будет получен один и тот же ответ, если используется автономный пакет ДДТ или ДДТ с разделением времени. Другими словами, возможен следующий диалог между программистом и ДДТ:

234/ 1234
234/ 1234

Поскольку эти виды ДДТ и прикладная программа не могут работать друг с другом в многопрограммном режиме, программа не выполняется, пока ДДТ ожидает входного сообщения от программиста. Таким образом, ячейка 234 будет с гарантией иметь одно и то же содержимое при двух последовательных опросах.

При использовании ДДТ реального времени, с другой стороны, *возможно,* что содержимое ячейки памяти внезапно изменится: причина, как объяснялось выше, в том, что проблемная программа продолжает выполняться, пока ДДТ реального времени ожидает входного сообщения от программиста. Таким образом, возможен был бы следующий диалог:

234/ 1234
234/ 1235
234/ 1236

После того как ДДТ напечатал содержимое ячейки памяти, говорят, что эта ячейка *открыта,* или предоставлена для изменения программисту. В этот момент он может либо изменить содержимое ячейки памяти, либо показать, что он удовлетворен текущим содержимым. Изменить содержимое ячейки памяти 234, например, программист мог бы с помощью следующего диалога:

234/ 1234 4321 <CR>

где символ <CR> обозначает «возврат каретки» или «конец строки». При более совершенной версии ДДТ возможен был бы следующий диалог:

XYZ + 3/ ADD ABC+1 SUBABC+1<CR>

Если программист удовлетворен содержимым ячейки памяти, он сообщает об этом символом <CR>, так что получается диалог следующего вида:

234/ 1234<CR>

ИЛИ

XYZ+3/ ADD ABC+1<CR>

После того как программист напечатал одну из вышеуказанных директив (либо изменяя содержимое ячейки памяти, либо оставляя его неизменным), говорят, что ячейка *закрыта,* т. е. она больше не может быть подвергнута изменению без другой явной директивы опроса.

Возможность опроса и изменения содержимого ячеек памяти является, безусловно, наиболее общей возможностью, свойственной самым различным версиям ДДТ; существует, однако, почти бесконечное число вариантов директив. Например, базовые директивы «открыть» и «закрыть» могут быть дополнены следующими директивами:

1. Открыть ячейку памяти без печати ее текущего содержимого. Полезна при изменении больших секций программы.
2. Закрыть текущую ячейку памяти (с изменением или без изменения содержимого) и одновременно открыть следующую соседнюю ячейку.
3. Закрыть текущую ячейку памяти и одновременно открыть предыдущую ячейку.
4. Закрыть текущую ячейку памяти и одновременно открыть ячейку, указанную исполнительным адресом, сформированным по содержимому текущей ячейки памяти.
5. Отпечатать содержимое ячеек памяти для заданного диапазона адресов, но без внесения изменений. Полезна для быстрого просмотра ряда ячеек памяти.

8.4.4. Директивы ДДТ для включения и удаления контрольных остановов

Крайне желательно, чтобы программист имел возможность остановить свою программу в середине прогона, когда она доходит до заданной ячейки памяти или команды. При работе с ДДТ это осуществляется с помощью директивы *«контрольный останов»* (breakpoint, B), которую выдает программист перед передачей управления своей проблемной программе. Так, он мог бы напечатать директиву

как указание пакету ДДТ остановить программу, если только управление будет передано в ячейку 456. В более совершенной версии

ДДТ такая директива могла бы иметь следующий вид:

```
XYZ-3$B
```

(символ $ необходим для того, чтобы отличить директиву контрольного останова от другого символического входного сообщения).

В некоторых версиях ДДТ программисту предоставляется возможность вставить до 16 различных контрольных остановов, каждому из которых присваивается свой номер. Так, он мог бы сказать: «Включить первый контрольный останов на ячейке 203», напечатав директиву

```
203$1B
```

Аналогично он мог бы потребовать: «Включить второй контрольный останов на ячейке 756», напечатав директиву

```
756$2B
```

Если программисту безразлично, какой по номеру контрольный останов используется, он мог бы сказать: «Включить следующий свободный контрольный останов на ячейке 404», напечатав директиву

```
404$B
```

Многочисленные контрольные остановы — исключительно эффективный инструмент программиста, если он ими правильно пользуется. Встретившись с ситуацией, где он определенно не знает, какое направление изберет его программа, программист может щедрой рукой разбросать контрольные остановы по всем возможным частям программы.

После вставки контрольных остановов программист обычно дает указание ДДТ передать управление его программе (с помощью директивы GO, о которой пойдет речь в подразд. 8.4.6). Если в процессе выполнения его программа доходит до одного из заданных контрольных остановов, управление возвращается ДДТ *до того, как будет выполнена команда,* хранящаяся в данной ячейке, и печатается соответствующее сообщение, помогающее программисту разобраться в ситуации. Так, в случае контрольного останова на ячейке 756 могло бы появиться следующее сообщение:

```
$2B  AT  LOC  756
```

Это сообщение, хотя оно выглядит несколько непонятно для непосвященного, содержит достаточную для программиста информацию; оно говорит о том, что программа дошла до второго заданного ей контрольного останова и остановилась непосредственно перед выполнением команды, хранящейся в ячейке 756. Некоторые версии ДДТ автоматически печатают содержимое сумматоров и (или) команду программы в ячейке контрольного останова, когда такйй останов происходит.

Директивы контрольных остановов выполняют одинаковые функции в автономных ДДТ и в ДДТ с разделением времени, но могут быть довольно сложными в ДДТ реального времени. Напомним, что тогда, когда программист ведет диалог с автономным ДДТ или ДДТ с разделением времени, его прикладная программа не выполняется. Она начинает реально выполняться только тогда, когда программист дает ДДТ специальную директиву передать управление его программе. Таким образом, пока он ведет диалог с ДДТ, он может вставить какой-то контрольный останов, а потом передумать и перенести его в другое место; аналогично, когда программа доходит до контрольного останова, ее выполнение прекращается и управление возвращается ДДТ.

При работе с ДДТ реального времени мы должны помнить, что прикладная программа не останавливается, когда программист ведет диалог с ДДТ. Таким образом, возможно, что, пока программист печатает директиву о включении контрольного останова в конкретную программу, эта программа продолжает выполняться. Более того, возможно, что программа подойдет к контрольному останову через мгновение после того, как ДДТ включил его; таким образом, программист может напечатать директиву контрольного останова и немедленнополучить сообщение, говорящее о том, что этот останов произошел.

Не всегда ясно, что должен делать ДДТ, когда происходит такой останов. С одной стороны, предполагается, что ДДТ реального времени позволяет прикладной программе выполняться, когда идет его диалог с программистом; в то же время назначение контрольного останова, с точки зрения программиста, состоит в том, чтобы *прекратить всю работу,* когда программа доходит до контрольного останова. Это значит, что ДДТ реального времени должен немедленно запретить все аппаратные прерывания (или установить для себя высший приоритет в системе), когда происходит контрольный останов. Когда программист печатает директиву ПРОДОЛЖИТЬ (которая рассматривается в следующем подразделе), ДДТ мог бы снова разрешить прерывания (или восстановить свой первоначальный уровень приоритета).

Предполагается, что немедленно после контрольного останова прикладная программа застывает на месте, находясь «в состоянии боевой готовности». Если это осуществимо, то такой режим исключительно полезен, поскольку он позволяет программисту точно узнать, что делала его программа в условиях реального времени в момент контрольного останова. К сожалению, это не всегда осуществимо. В системе управления реальными процессами, например, подобный контрольный останов мог бы вызвать аварийную ситуацию.

Наконец, требуется директива, с помощью которой можно было бы изъять один или несколько контрольных остановов проблемной

программы. Для простой версии ДДТ, рассчитанной только на один контрольный останов, достаточно было бы директивы

```
B
```

При использовании более совершенной версии программист должен был бы напечатать

```
$3B
```

чтобы исключить третий контрольный останов программы, и

```
$B
```

чтобы исключить *все* остановы.

8.4.5. Директива ДДТ для продолжения выполнения программы после контрольного останова

После того как произошел контрольный останов, программист с помощью ДДТ просматривает соответствующие переменные и ячейки памяти, производит необходимые исправления и в общем оценивает прохождение программы до данной точки. Убедившись, что программа идет правильно, он хочет обычно продолжить ее выполнение. Такую возможность ему предоставляет директива ПРОДОЛЖИТЬ (proceed) пакета ДДТ; ДДТ принимает эту директиву, когда программист печатает ее символическое обозначение

```
P
```

В этот момент возобновляется выполнение прикладной программы с точки последнего контрольного останова.

Некоторые версии ДДТ предусматривают возможность задания так называемых условных контрольных остановов и директив продолжения счета. Наиболее распространенная форма директивы условного продолжения счета дает указание ДДТ не останавливаться на текущей контрольной точке повторно до тех пор, пока программа не пройдет через нее определенное число раз. В качестве примера предположим, что программист пытается отладить секцию своей программы, где 100 раз выполняется один и тот же цикл. Он мог бы пожелать включить контрольный останов, чтобы проверить, правильно ли выполняется программа при начальном проходе через этот цикл; затем он хотел бы снова остановить программу на последнем проходе через цикл. Когда этот контрольный останов происходит первый раз, программист может напечатать

```
98P
```

как указание ДДТ не останавливаться в этой контрольной точке, пока программа не пройдет через нее еще 98 раз. При 99-м проходе через цикл (который в действительности является сотым проходом,

считая первоначальный) ДДТ остановит программу в этой контрольной точке и передаст управление программисту.

В версии ДДТ с исключительно широкими возможностями директива условного продолжения счета по своим функциям аналогична *условной трассировке,* рассмотренной выше в этой главе. Другими словами, программисту предоставляется возможность указать, что контрольный останов не следует принимать во внимание до тех пор, пока, например, значение содержимого сумматора не станет равным нулю, в ячейке памяти 456 не окажется число 7654, а в индекс-регистре 3 — число минус 7. Столь широкие возможности ДДТ — дело почти неслыханное, однако их не так уж трудно реализовать.

8.4.6. Директива ДДТ для передачи управления прикладной программе

Чтобы передать управление своей программе, программист обычно печатает директиву следующего вида:

```
567G
```

или

```
START + 3SG
```

8.4.7. Директивы ДДТ для организации операций поиска

Другие возможности ДДТ реализуются при возникновении соответствующей потребности, однако многие из них весьма распространены в различных реализациях и оказываются довольно удобными. К числу наиболее удобных возможностей относятся директивы поиска, предусматриваемые во многих пакетах ДДТ. При отладке программисту часто желательно выявлять все обращения к конкретной команде, все случаи использования некоторой константы, все обращения к определенному индекс-регистру и т. д. Это можно сделать с помощью операций *поиска по слову (по содержимому), по отсутствию* и *по исполнительному адресу.*

Предположим, что программист хочет просмотреть свою программу, чтобы найти все ячейки, где встречается константа 4567; чтобы сделать это, он печатает директиву

```
1000, 2000, 4567W
```

Эта директива является для ДДТ командой просмотра программы от ячейки 1000 до ячейки 2000 с распечаткой содержимого всех ячеек памяти, где хранится константа 4567 (на первый взгляд, распечатка содержимого всех этих ячеек может воказаться излишней, но вскоре станет ясно, что это оправданно). Так, ДДТ мог бы по получении вышеуказанной директивы отпечатать следующую

информацию:

```
1234/        4567
1456/        4567
1457/        4567
1776/        4567
```

При более совершенной версии ДДТ программист мог бы напечатать на символическом языке директиву следующего вида:

START, END, CAT + DOG$W

Здесь опять-таки задаются те же самые три параметра: нижняя и верхняя границы поиска и искомый элемент.

Предположим теперь, что программист хочет просмотреть некоторую таблицу в поисках всех ненулевых значений. В этом случае он производит поиск не по наличию конкретной величины, а по ее *отсутствию,* а именно по отсутствию нулевого значения.

Чтобы задать такой поиск, программист печатает

1000, 2000, ON

Для ДДТ это является указанием просмотреть память от ячейки 1000 до ячейки 2000 и найти все ячейки с ненулевым содержимым. В результате может быть напечатана следующая информация:

```
1234/        4567
1324/        7654
1605/        7777
1776/        OOIII
```

Наконец, программисту может потребоваться просмотреть программу для определения всех обращений к конкретной ячейке. Такую возможность иногда дают таблицы перекрестных ссылок в конце листинга программы на языке ассемблера, однако ДДТ обеспечивает полное вычисление исполнительных адресов (в том числе при косвенной адресации, индексации, адресации с базовым адресом и т. д.) в *процессе выполнения программы,* чего ассемблер делать не может.

Предположим, например, что программисту нужно найти все команды, обращающиеся к ячейке XYZ+3. При совершенной версии ДДТ он мог бы напечатать следующую директиву:

START, END, XYZ + 3$E

в результате выполнения которой от ДДТ был бы получен следующий ответ:

START + 4/ ADD XYZ + 3
LOOP+1/ SUB XYZ + 3
LOOP + 6/ STORE XYZ, 3 (ОБРАЩЕНИЕ СВЯЗАНО С ИНДЕКСАЦИЕЙ)
ABC+14/ LOAD*PNTR (ОБРАЩЕНИЕ СВЯЗАНО С КОСВЕННОЙ АДРЕСАЦИЕЙ)

Многие версии ДДТ предусматривают различные команды поиска с использованием *маски.* В этом случае ДДТ, проверяя некоторое слово памяти на соответствие условию поиска, анализирует только те биты, которые указаны маской. При отсутствии специальных указаний ДДТ устанавливает значения всех битов маски равными единице, разрешая таким образом поиск по признаку совпадения или несовпадения с полным словом. Устанавливая маску с помощью отдельнойдирективы ДДТ, программист может анализировать любую комбинацию битов слова.

Предположим, например, что программист хочет найти все обращения к индекс-регистру 3. Для данного примера примем, что программист работает с 24-битовой машиной, в которой номер индекс-регистра указывается битами 6—8 команды. Поэтому программист мог бы установить маску следующим образом:

```
00700000M
```

Эта маска является для ДДТ указанием того, что при любой из директив поиска анализ следует ограничить только битами 6, 7 и 8. Сам поиск можно было бы задать следующей директивой:

```
1000,2000,11311111W
```

той же цели могла бы послужить следующая директива:

```
1000,2000,777377777W
```

8.5. Реализация простой версии ДДТ

Завершая обсуждение ДДТ, мы представим здесь краткое описание реализации простой версии ДДТ. Этот пакет ДДТ предназначается для машины, ориентированной на пословную обработку данных и будет воспринимать только восьмеричные входные коды. Кроме того, он позволяет включать только один контрольный останов. Однако такая простота имеет свои преимущества: представленный здесь ДДТ можно реализовать с помощью всего 200 команд (или даже меньше) и закодировать за несколько часов.

Входное сообщение для этой простой версии ДДТ состоит из восьмеричного числа, вслед за которым вводится командный символ; может быть также, что в качестве входного сообщения высту-

пает один командный символ. Такимобразом, допустимое входное сообщение ДДТ может иметь следующий вид:

```
1234/
1234G
4321B
B
P
```

Программа ДДТ при работе просто последовательно накапливает восьмеричное число, пока не обнаружит командный символ; затем она выполняет действие, заданное этим командным символом.

8.5.1. Программа инициализации ДДТ

На рис. 8.5 показана программа инициализации, которая по сути довольно проста. Вначале она печатает символ возврата каретки (или какое-либо другое соответствующее сообщение), давая знать программисту, что она готова к работе и ждет входного сообщения. Затем инициализируются три флажка и один внутренний рабочий

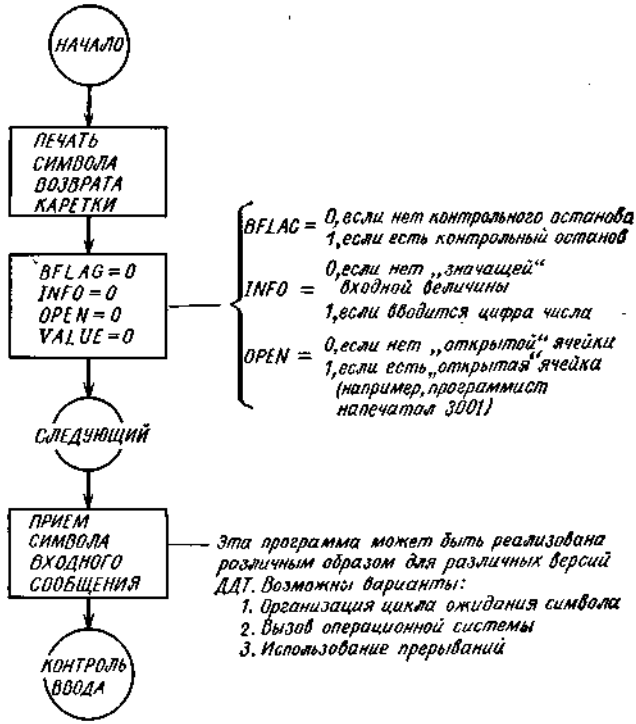


Рис. 8.5. Инициализация и основной цикл.

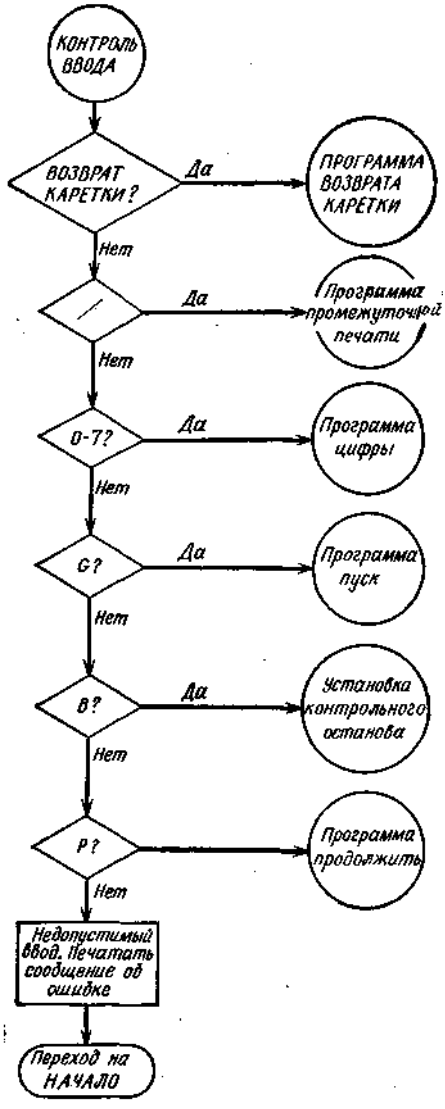


Рис. 8.6. Программа анализа символа входного сообщения.

регистр: BFLAG (КФЛАЖОК, флажок контрольного останова), INFO (ИНФО, флажок ввода цифровой информации), OPEN (ОТКР, флажок открытой ячейки памяти) и VALUE (ЗНАЧЕНИЕ).

BFLAG — это признак, говорящий ДДТ о том, напечатал ли программист директиву контрольного останова. Если BFLAG=0,

контрольного останова нет; если BFLAG=1, то контрольный останов задан. В начале работы ДДТ BFLAG принимает нулевое исходное значение; когда программист печатает директиву «B», этот флажок устанавливается в единицу или сбрасывается в нуль.

INFO — это признак, говорящий о том, принимает ли ДДТ какую-либо числовую информацию в составе входного сообщения. INFO устанавливается в нуль в начале работы ДДТ и сбрасывается в нуль каждый раз, когда ДДТ начинает анализировать новое входное сообщение; он устанавливается в единицу, когда ДДТ принимает восьмеричную цифру, как это будет показано ниже. Таким образом, если программист печатает директиву

```
1234B
```

то к моменту, когда ДДТ будет обрабатывать символ «B», признак INFO будет установлен в единицу. С другой стороны, если программист печатает

```
B
```

то признак INFO будет нулевым, когда ДДТ примет символ «B».

OPEN — это признак, говорящий ДДТ о том, имеется ли открытая ячейка памяти. Если OPEN=0, то в данный момент нет открытой и доступной для изменения содержимого ячейки памяти; если OPEN=1, то в данный момент открыта ячейка памяти, адрес которой указан LOC, внутренней переменной ДДТ. Флажок OPEN устанавливается в нуль программой инициализации и устанавливается в единицу программой промежуточной печати (обработки символа косой черты).

Наконец, VALUE — внутренний рабочий регистр, где накапливается значение восьмеричной входной величины в процессе ее ввода. Величина VALUE устанавливается в нуль при инициализации и сбрасывается в нуль, когда ДДТ начинает анализировать новое входное сообщение.

После выполнения инициализации ДДТ переходит на основной цикл ввода, показанный в виде блок-схемы рис. 8.6 и описанный ниже.

8.5.2. Основной цикл ввода

Задача основного цикла ввода заключается в том, чтобы принять один символ входного сообщения с терминала программиста и решить, что делать дальше. Как мы отмечали выше, именно программа «прием символа» отличает автономный пакет ДДТ от ДДТ с разделением времени и ДДТ реального времени.

После приема символа входного сообщения ДДТ анализирует его, чтобы убедиться, что это один из допустимых входных символов: восьмеричная цифра, символ косой черты, возврата каретки, G,

В или Р. Если это так, ДДТ передает управление программе обработки соответствующего символа; если нет, управление переходит к программе ошибки.

8.5.3. Программа обработки восьмеричной цифры

Задача этой программы, как показывает блок-схема на рис. 8.7,—. принять одну восьмеричную цифру и сформировать (накопить) текущее содержимое регистра VALUE. Программа также устанавливает в единицу флажок INFO, чтобы ДДТ мог впоследствии «вспомнить», сопровождалась ли директива восьмеричным числом.

Программа работает чрезвычайно просто. Она умножает текущее содержимое регистра VALUE на восемь (что в большинстве машин может быть сделано путем сдвига) и прибавляет новую восьмеричную цифру. В блок-схеме рис. 8.7 не предусмотрен контроль ошибок. Если программист вводит слишком большое число, регистр VALUE просто переполняется, причем никакого признака ошибки не выдается. Расширить программу, добавив подобный контроль ошибок, не составляет труда; аналогично программу можно легко доработать, чтобы иметь возможность ввода десятичных или шестнадцатеричных чисел.

Рис. 8.7. Программа обработки цифры.

8.5.4. Программа промежуточной печати

Как мы видели на рис. 8.6, программа промежуточной печати вступает в действие, когда основной цикл ввода обнаруживает символ. В этот момент ясно, что программист, должно быть, напечатал что-то типа

1234/

и задача программы промежуточной печати, как это показано на рис. 8.8,— напечатать на терминале содержимое ячейки памяти 1234.

Вначале, однако, ДДТ должен запомнить текущее содержимое VALUE. После того как будет отпечатано содержимое указанной ячейки памяти, ДДТ вернется на основной цикл ввода, ожидая дополнительных инструкций от программиста; следовательно, VALUE придется сбросить в нуль. Поэтому текущее содержимое VALUE, представляющее собой адрес опрашиваемой ячейки памяти, сохраняется в LOC (ЯЧ).

Затем ДДТ печатает текущее содержимое ячейки памяти, формируя выходное сообщение таким образом, чтобы его можно легко

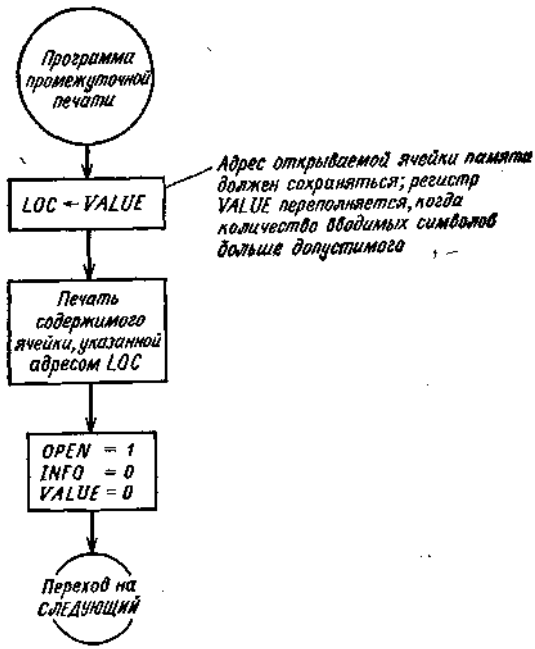


Рис. 8.8. Программа промежуточной печати.

отличить:

1234/ 7654

В этот момент ДДТ не знает, будет ли программист менять содержимое ячейки или нет; поэтому ячейка считается открытой и ДДТ ожидает дополнительных сообщений от программиста.

Перед возвращением на основной цикл ввода ДДТ устанавливает флажок OPEN в единицу, чтобы помнить о наличии открытой ячейки памяти. Флажок INFO сбрасывается в нуль, так что ДДТ сможет узнать, будет ли следующая директива содержать какие-либо восьмеричные цифры; VALUE также сбрасывается в нуль.

8.5.5. Программа возврата каретки

Основное назначение сигнала возврата каретки — закрыть открытую ячейку памяти. Это может делаться с изменением или без изменения текущего содержимого данной ячейки.

Программа возврата каретки, показанная на рис. 8.9, вначале анализирует значение флажка INFO. Если это нуль, то программист,

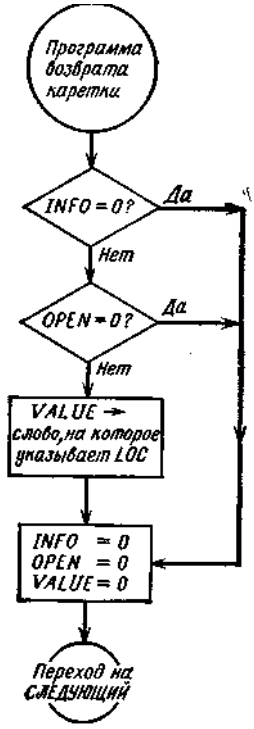


Рис. 8.9. Программа возврата каретки.

по-видимому, напечатал входное сообщение следующего вида:

1234/7654<CR>

Другими словами, если флажок INFO=0, то непосредственно перед символом возврата каретки никакие восьмеричные цифры не вводились. Это свидетельствует о том, что программист не хотел изменять содержимое ячейки памяти; после этого ДДТ сбрасывает флажок OPEN и регистр VALUE и возвращается на цикл ожидания ввода.

Если флажок INFO установлен в единицу, это говорит о том, что программист напечатал директиву следующего вида:

1234/ 7654 4567 <CR>

То есть, если флажок INFO=1, это показывает, что непосредственно перед программой возврата каретки имел место ввод восьмеричного числа. Это восьмеричное число накоплено в регистре VALUE программой обработки восьмеричных цифр и представляет собой то измененное содержимое, которое программист хочет занести в ячейку памяти, указанную адресом LOC.

Однако, прежде чем произвести указанное изменение, программа ДДТ проверяет, установлен ли в единицу флажок OPEN. Тем самым не допускается, чтобы программист печатал директиву вида

1234<CR>

и предотвращается изменение содержимого ячейки памяти без предварительного его просмотра. Аналогичные виды средств контроля ошибок часто встраиваются и в другие программы ДДТ, но здесь речь идет о настоятельной необходимости: ошибка со стороны программиста может вызвать порчу части его прикладной программы.

Если все правила соблюдены, ДДТ меняет содержимое ячейки памяти по указаниюпрограммиста. Затем ДДТ сбрасывает флажок INFO, флажок OPEN и регистр VALUE; наконец, ДДТ возвращается на основной цикл ввода.

8.5.6. Программа ПУСК

Назначение программы GO (ПУСК), показанной на рис. 8.10, состоит в том, чтобы передать управление от ДДТ отлаживаемой программе. Адрес, по которому передается управление, указывается программистом в его директиве

1234G

На всякий случай программа ПУСК должна вначале проанализировать флажок INFO, чтобы удостовериться, что программист не

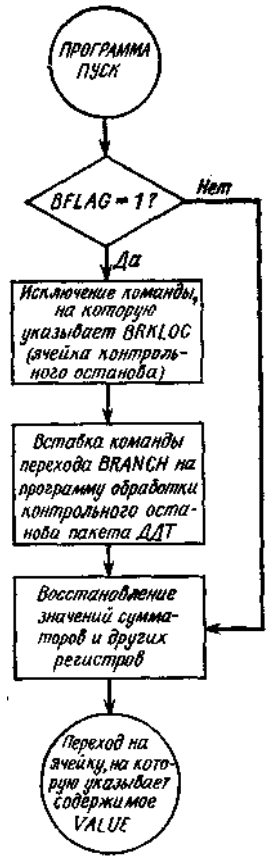


Рис. 8.10. Программа ПУСК.

напечатал только

G

Для простоты в нашей версии ДДТ это не предусмотрено; подобная ошибка программиста вызвала бы передачу управления от ДДТ к нулевой ячейке. При таком подходе лучше было бы, по-видимому, если бы ДДТ выдавал программисту предупреждающее начальное сообщение «Caveat emptor».

Главная задача ДДТ — проверить, включил ли программист заблаговременно контрольный останов. Как показано на рис. 8.10, это выполняется путем анализа признака BFLAG. Если он установлен в единицу, ДДТ удаляет содержимое той ячейки памяти, в которой программист задал выполнение контрольного останова; в эту

ячейку помещается специальная команда, вызывающая переход программы на программу обработки контрольного останова (рис. 8.12).

Все сумматоры, индекс-регистры или прочие слова состояния, которые, как ожидает прикладная программа, должны быть настроены соответствующим образом, затем восстанавливаются ДДТ. Наконец, управление передается в ячейку, заданную программистом; как обычно, ее адрес содержится в регистре VALUE.

8.5.7. Программа включения контрольного останова

Зта программа, показанная на рис. 8.11, делает не очень много; ее основная функция — запомнить, что программист дал задание на контрольный останов, чтобы этот останов можно было включить впоследствии, когда будет обрабатываться директива ПУСК.

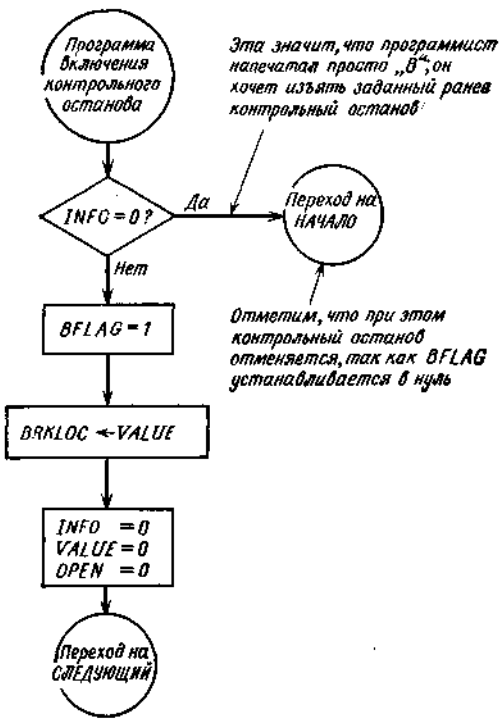


Рис. 8.11. Программа включения контрольного останова.

Вначале программа анализирует флажок INFO, чтобы установить, имела ли директива вид

```
1234B
```

или просто

```
B
```

Как и в случае предыдущих программ, если INFO=0, то перед символом «B» никакого восьмеричного числа не было; если 1NFO=1, то программа контрольного останова знает, что директива состояла из восьмеричного числа с последующим символом «B».

Если программист печатает только символ «B», то предполагается, что он хочет изъять заданный им контрольный останов. ДДТ это легко осуществить, если возвратиться на начало программы инициализации, где все флажки, в том числе BFLAG сбрасываются.

Если INFO=1, то это означает, что программист просит, чтобы был включен контрольный останов на конкретной ячейке. В этот момент ДДТ просто устанавливает BFLAG в единицу, чтобы помнить, что задан контрольный останов; ДДТ запоминает ячейку контрольного останова (адрес которой, как обычно, хранится в регистре VALUE) как внутреннюю переменную под названием BRKLOC (ячейка контрольного останова). Затем ДДТ сбрасывает флажок INFO, регистр VALUE и возвращается на основной цикл ввода.

Здесь, быть может, следует объяснить, почему ДДТ не заносит свою специальную команду перехода сразу же, как только программист выдаст директиву контрольного останова. Причину легко понять, если посмотреть на типичную последовательность действий программиста:

```
1234 B
1234/   7654
1235B
```

В этом примере видно, что программист решил включить контрольный останов на ячейке 1234. Однако немного поразмыслив, он усомнился, в нужном ли месте он задал останов, и попросил посмотреть ячейку 1234. Если бы ДДТ немедленно записал в ячейку 1234 свою команду перехода, программист увидел бы уже *эту команду* вместо первоначального содержимого ячейки и был бы, естественно, озадачен.

Вследствие этого в ДДТ принято включать контрольный останов только в момент, когда управление передается проблемной программе, т. е. когда программист печатает директиву «G» (ПУСК) или «P» (ПРОДОЛЖИТЬ).

8.5.8. Программа обработки контрольного останова

Программа обработки контрольного останова, показанная на рис. 8.12, вступает в работу, когда прикладная программа доходит до ранее заданного контрольного останова. Вместо выполнения команды, которая, как считает программист, находится в данной ячейке, программа передает управление программе обработки контрольного останова, входящей в ДДТ.

Первая задача ДДТ — сохранить значения всех сумматоров, индекс-регистров и прочих оперативных регистров, чтобы в дальнейшем прерывание никак не отразилось на ходе прикладной программы. Эти регистры восстанавливаются, когда печатается директива «G» или «P».

Затем программа обработки контрольного останова печатает сообщение программисту, уведомляя его, что произошел контрольный останов. Сообщение в данной версии ДДТ может быть тривиальным, поскольку она предусматривает всего один контрольный останов.

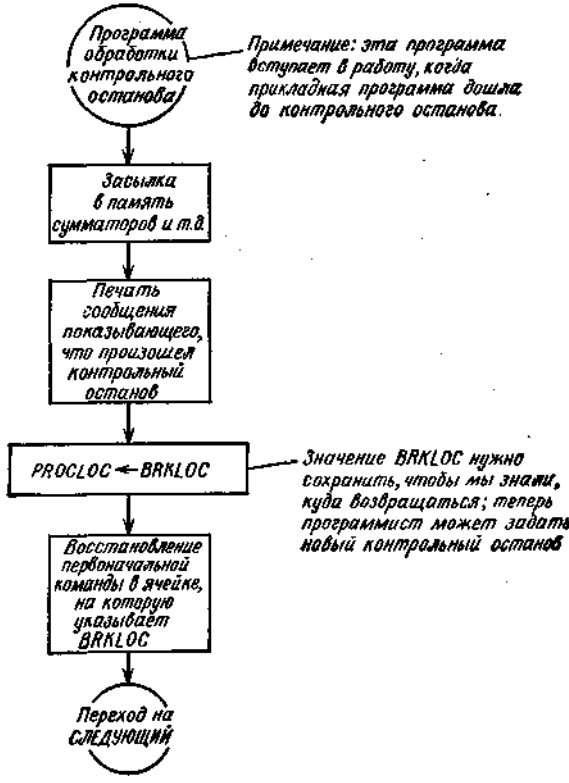


Рис. 8.12. Программа обработки контрольного останова.

После этого ДДТ должен запомнить тот факт, что прикладная программа прервана на ячейке контрольного останова. Как было показано на рис. 8.11, адрес этой ячейки запоминается в BRKLOC; содержимое BRKLOC теперь переписывается как другая внутренняя переменная ДДТ под названием PROCLOC. Это необходимо, поскольку программист мог бы, например, напечатать следующую последовательность входных директив:

```
l234B
4321G
[СООБЩЕНИЕ КОНТРОЛЬНОГО ОСТАНОВА]
2003B
P
```

То есть после первого контрольного останова программист мог решить переместить контрольный останов на другую ячейку. Таким образом, хотя значение BRKLOC могло и измениться, *ДДТ необходимо помнить старое значение для продолжения выполнения прикладной программы.*

Затем ДДТ восстанавливает команду, которая ранее хранилась в ячейке контрольного останова. Благодаря этому, отметим еще раз, программист может просматривать свою программу, не встречая никаких специальных команд, команд перехода ДДТ.

8.5.9. Программа ПРОДОЛЖИТЬ

Программа ПРОДОЛЖИТЬ, показанная на рис. 8.13, очень похожа на программу ПУСК, рассмотренную выше. Вначале она проверяет, существует ли контрольный останов, и, если да, вставляет специальную команду перехода и запоминает первоначальную команду прикладной программы. Затем она восстанавливает содержимое сумматоров, индекс-регистров и т. д. в том виде, что был в момент последнего контрольного останова.

Наконец, ДДТ нужно возвратить управление прикладной программе. Однако здесь в принципе возможно осложнение, поскольку на той ячейке, к которой нужно возвратиться, может быть задан контрольный останов. Предположим, например, что программист печатает следующую последовательность директив:

```
1234B
4321G
[СООБЩЕНИЕ КОНТРОЛЬНОГО ОСТАНОВА]
P
```

Контрольный останов произошел на ячейке 1234 непосредственно перед выполнением команды, поскольку ДДТ записал, без ведома программиста, свою специальную команду в ячейку 1234. Когда программист печатает «P», он полагает, что теперь выполнится его

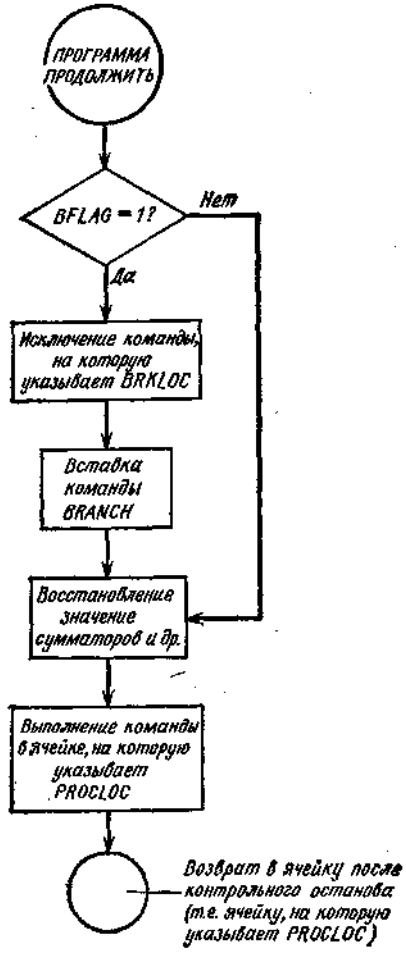


Рис. 8.13. Программа ПРОДОЛЖИТЬ.

команда, хранящаяся в ячейке 1234, и что программа будет продолжать выполняться до тех пор, пока снова не подойдет к ячейке 1234, где снова произойдет контрольный останов.

Это означает, что ДДТ должен исполнить команду, которую программист первоначально заносил в ячейку 1234, а затем передать управление в ячейку 1235. Таким образом, ДДТ удастся сохранить свою специальную команду перехода в ячейке 1234. Это довольно легко сделать на любой машине, где имеется команда EXE (исполнить). Последовательность команд в конце программы ПРОДОЛ-

ЖИТЬ пакета ДДТ в этом случае могла бы выглядеть так:

```
EXE       SAVLOC
BRANCH    1235
BRANCH    1236
```

Если первоначальная команда ячейки 1234 (которая, как предполагается, запоминается в ячейке SAVLOC) — это простая команда, например сложение, то управление будет передано команде, следующей за командой EXE, т. е. команде перехода BRANCH 1235. Если же команда в SAVLOC — это команда *пропуска* того или иного типа, то будет выполнена команда BRANCH 1236 и прикладная программа возьмет управление на себя.

Отметим, что ситуация с контрольным остановом может оказаться довольно сложной. Если в машине нет команды *исполнить,* ДДТ придется запоминать первоначальную команду прямо в собственной последовательности команд, т. е. соответствующий участок программы ПРОДОЛЖИТЬ будет иметь вид

```
SAVLOC    **
          BRANCH    1235
          BRANCH    1236
```

Если программист задает контрольный останов на команде вызова подпрограммы, то также могут возникнуть сложности. При выполнении команды вызова подпрограммы адрес *возврата* сформируется *внутри* ДДТ, а не внутри прикладной программы. Этот вопрос можно решить двумя способами: либо считать недопустимым задание контрольного останова на команде вызова подпрограммы, либо включить в программу ПРОДОЛЖИТЬ средства опознавания и *средства программного моделирования подобной команды.*

Отметим, наконец, что в случае, если программист задает контрольный останов на ячейке, содержимое которой программа в ходе работы меняет, получается полная неразбериха.

ЛИТЕРАТУРА

1. Yourdon E. (ed.), Real-Time Systems Design, Information and Systems Press, Cambridge, Mass., 1967.
2. Boilen S., et al., A Time-Sharing Dedugging System for a Small Computer, Proceedings, Spring Joint Computer Conference, 1963.
3. Evans T. G., Darley D. L., DEBUG—An Extension to Current On-Line Debugging Techniques, Communications of the ACM, v. 8, № 5, May 1965.
4. Van Steeg R. L., Talk—A High Level Source Language Debugging Technique, Communications of the ACM, v. 7, № 7, July 1964.
5. Rustin R. (ed.), Debugging Techniques in Large Systems, Prentice-Hall, Inc., 1971.
6. Gaines R. S., The Debugging of Computer Programs, Ph. D. Thesis, Princeton University, 1969.
7. Yourdon E., Design of On-Line Computer Systems, Prentice-Hall, Inc., 1972, ch. W.

ВОПРОСЫ

1. Дайте определение термина «отладка». Чем отладка отличается от тестирования? Считаете ли вы отладку искусством или наукой? Как вы думаете, почему одни программисты с легкостью отлаживают свои программы, а другие — нет?

2. Почему ошибки легче находить, если программа разбита на небольшие модули? Уверены ли вы в этом в действительности? Считаете ли вы, что при этом в программу будет, быть может, вноситься больше ошибок из-за проблем интерфейса модулей? Что следовало бы сделать, чтобы этого избежать?

3. Как действует программист, пытаясь определить природу ошибки (см. подразд. 8.1.2)? Считаете ли вы, что это умение, которым легко могут овладеть все программисты?

4. Почему целесообразно проверять, является ли ошибка повторяющейся и постоянной? Приведите какие-либо иримеры причин, по которым ошибка может не быть повторяющейся.

5. Приведите пример исчерпывающего, методичного и логичного набора процедур поиска ошибок. Можно ли применять ваш набор процедур для большинства программ, над которыми вы работаете?

6. Почему целесообразно при поиске ошибки в программе начинать с проверки простейших предположений?

7. В чем смысл разговора с коллегой об ошибке в вашей программе? Считаете ли вы, что имеет значение, с кем из коллег вы говорите? Есть ли у вас коллеги-программисты, с кем вам особенно приятно и удобно работать при отладке программы?

8. Приведите пример тестового режима, который вы создаете для получения подтверждений своим подозрениям относительно некоторой ошибки в программе? Часто ли можно применять такой подход?

9. Почему часто бывает полезно «отключиться» от какой-либо сложной проблемы отладки до следующего дня?

10. Можете вы предложить какие-либо другие принципы или стратегии отладки?

11. Приведите пример ошибки перфорирования или переписки, которая привела бы к неработоспособности одной из ваших программ? Часто ли возникает такая ситуация? Можно ли легко избежать ее?

12. Приведите пример программной ошибки, вызванной отсутствием инициализации переменной в одной из ваших программ. Как можно избежать подобной ошибки? Как можете вы определить, когда подобная ошибка случилась?

13. Приведите пример программной ошибки, вызванной совместным использованием переменных или рабочей памяти многими модулями. Как можно было бы избежать такой ошибки? Как можно определить, когда подобная ошибка возникла, т. е. имеет ли она какие-либо типичные симптомы?

14. Приведите пример программного дефекта вследствие ошибок управления или логики, например вызов модулей, когда этого не должно было бы быть, и г. д. Как можно было бы легко обнаружить подобную ошибку?

15. Приведите пример программной ошибки, вызванной индексацией с выходом за границы массива. Часто ли встречается такая ошибка в ваших программах? Как ее можно избежать? Как можете вы определить, есть ли ошибка такого рода в вашей программе?

16. Приведите пример программной ошибки, вызванной неправильным окончанием последовательностей команд программы. Часто ли возникает подобная проблема в ваших программах? Как ее можно избежать? Как можно ее распознать, когда она встречается?

17. Приведите пример ошибки, вызванной тем, что не предусмотрена обработка особых случаев. Была ли это действительно программная ошибка или это был недостаток технического задания на программу?

18. Приведите пример ошибки, вызванной неверными связями и интерфейсами между модулями. Типичноли это для ваших программных проектов? Как этого

можно избежать? В какой степени могут быть здесь полезны принципы программирования с защитой от ошибок, рассмотренные в гл. 6?

19. Приведите пример ошибки, вызванной неверным выбором форматов данных в одной из ваших прикладных программ на языке высокого уровня. Помогает ли ваш компилятор легко обнаруживать подобные ошибки? Каким еще способом можно их обнаруживать?

20. Приведите пример программной ошибки, вызванной передачей подпрограмме констант в качестве параметров. Часто ли это случается? Есть ли какой-либо способ распознавания такой ошибки, когда она происходит?

21. Приведите пример ошибки, вызванной неправильным использованием сложных булевых выражений в одной из ваших программ. Как можно избежать подобных ошибок? Как можно их распознать, когда они возникают? Считаете ли вы, что в некоторых языках программирования подобные ошибки возникают более часто, чем вдругих?Считаетели вы, чтоони возникают более часто в тех случаях, когда программисты получают определенную подготовку или, напротив, не получают ее?

22. Приведите пример ошибки, вызванной неправильным использованием вложенных операторов IF-THEN-ELSE в одной из ваших программ. Типичная ли это ошибка? Считаете ли вы, что должно быть, по-видимому, больше таких ошибок при применении методов структурного программирования, подобных рассмотренным в гл. 4? Как можно легко распознать такие ошибки?

23. Приведите пример программной ошибки, вызванной некорректным выходом из подпрограммы в одной из ваших программ. Типичная ли это ошибка? Как ее можно распознать, когда она встречается?

24. Приведите пример ошибки, вызванной обращением к областям данных после выполнения оператора WRITE. Типичная ли это ошибка? Как ее можно избежать? Как можно ее распознать, когда она встречается?

25. Приведите пример ошибки, вызванной неправильным использованием флажков в одной из ваших программ. Типичная ли это ошибка? Происходит ли она чаще в программах пакетной обработки или в программах реального времени? Как можно ее избежать? Как можно ее распознать, когда она возникает?

26. Приведите пример ошибки, вызванной тем, что не предусмотрена обработка особых случаев ввода-вывода в одной из ваших программ. Типичная ли это ошибка? Как можно ее распознать, когда она возникает?

27. Приведите пример программной ошибки, вызванной отсутствием анализа кодов ответа. Встречается ли эта проблема более часто, когда коды ответа формируются операционной системой фирмы-изготовителя или пакетом программных средств фирмы-поставщика? Как ее можно избежать? Как можно ее распознать, когда она встречается?

28. Приведите пример программной ошибки, вызванной тем, что счетчики были слишком малой разрядности, и со временем произошло их переполнение. Можно ли легко избежать такой ошибки? Как можно ее распознать, когда она возникает?

29. Приведите пример программной ошибки, обусловленной в свою очередь ошибкой в сложном выражении в одной из ваших программ. Часто ли встречается ошибка этого рода? Вероятнее ли она для научно-технических или экономических приложений? Как можно ее распознать, когда она возникает?

30. Приведите пример программной ошибки, вызванной операциями выравнивания? Типичная ли это ошибка? Вероятнее ли она для машин, работающих со словами фиксированной длины, или для машин, обрабатывающих данные в байтовом формате?

31. Приведите пример ошибки, вызванной выходом за границы уровня подпрограммы. Типичная ли это ошибка? Как можно ее распознать, когда она возникает?

32. Приведите пример программной ошибки, вызванной использованием ошибочных общих регистров или индекс-регистров. Типичная ли это ошибка для

вашей организации? Как можно ее избежать? Как можно ее распознать, когда она возникает?

33. Приведите пример ошибки, вызванной попытками работать сданными как с командами в одной из ваших программ. Как можно избежать подобной ошибки? Как можно ее распознать, когда она возникает?

34. Приведите пример ошибки, при анализе которой в конце концов обнаружилось, что использовалась неправильная версия программы. Часто ли встречается такая ситуация в вашей организации? Как можно ее предотвратить?

35. Приведите пример программной ошибки, вызванной отсутствием признака конца элемента программы (например, отсутствием точки с запятой). Считаете ли вы, что ошибка этого типа более вероятна для определенных типов языков? Как можно ее распознать, когда она возникает?

36. Каково основное назначение дампингов памяти? Назовите некоторые из причин, по которым дампинги сейчас не используются столь широко, как ранее.

37. Приведите пример программной ошибки по крайней мере одного типа из числа не упомянутых в разд. 8.2.

38. В чем основное различие между трассировками и дампингами? Когда рекомендуется применять трассировки? Когда рекомендуется применять дампинги?

39. Приведите пример эффективного использования «трассировки для подпрограммы» в случае отладки вашей программы.

40. Приведите пример эффективного использования «трассировкидля переменной» в случае отладки вашей программы.

41. Приведите пример эффективного использования «трассировки по времени». Часто ли встречается подобная ситуация?

42. Приведите пример эффективного использования «трассировки для терминала». Какие управляющие средства следовало бы предусмотреть, чтобы трассировка не вызывала слишком больших накладных расходов?

43. Приведите пример трассировки какого-либо другого типа, отличающегося от рассмотренных в разд. 8.3.

44. Каковы основные типы пакетов динамической отладки (ДДТ)? Когда их следует применять? Есть ли в вашей организации подобные пакеты программ? Поощряется ли их применение?

УПРАЖНЕНИЯ И ЗАДАЧИ ДЛЯ КЛАССНЫХ ЗАНЯТИЙ[1)]

Введение

Упражнения и задачи для классных занятий — организационные вопросы

А. ЦЕЛИ ПРЕПОДАВАТЕЛЯ

1. Главное назначение приводимых в приложении задач для решения на классных занятиях заключается в том, чтобы проиллюстрировать методы и концепции, изложенные в нашей книге. Даже принимая во внимание то, что некоторые примеры приводятся непосредственно в тексте самой книги, мы все же ощущаем необходимость дать некоторый обобщающий пример, в котором удалось бы использовать возможно большее число методов программирования.

2. Другая важная цель, которую мы преследуем при решении задач на классных занятиях, связана с тем, чтобы дать студенту возможность применить на практике методы и концепции, рассмотренные в нашей книге. Имеющийся опыт преподавания учебных курсов по программированию показывает, что многие понятия, излагаемые преподавателем в классе, не будут усвоены, а в некоторых случаях и не поняты до тех пор, пока они не будут использованы при решении реальных задач.

3. Были затрачены определенные усилия на выбор ряда задач, которые бы, с одной стороны, не были столь тривиальными, чтобы казаться скучными и утомительными, а с другой — были бы не очень сложными, поскольку время обучения весьма ограничено.

4. И наконец, мы надеемся, что приводимые здесь примеры дадут еще одну возможность и преподавателям, и студентам поэкспериментировать и таким образом немножко больше узнать об искусстве «черной магии», которое мы называем программированием. В первую очередь мы старались сосредоточить внимание на нашем методе решения задач; кроме того, мы старались дополнительно

[1)] Перевод выполнен Б. С. Овчинниковым.

привлечь внимание к таким концепциям программирования, как модульность, простота построения программ; к методам, исключающим необходимость отладки программ (концепции, рассматриваемые в гл. 5 и 6), к структурному программированию (гл. 4) и иерархическому построению программ (гл. 2).

Б. ВАШИ ЦЕЛИ

1. Основной целью учебных упражнений по программированию является иллюстрация концепции иерархического построения программы. Вы должны разработать и написать программу в виде последовательности из нескольких уровней, начиная с главной программы и продвигаясь вниз (в иерархическом смысле) до тех пор, пока не будут написаны самые подробные модули программы. Каждый модуль должен представлять собой законченную функциональную часть, имеющую определенное назначение и заданные начало и конец. Кроме того, по объему модуль должен быть достаточно малым.

2. Второе назначение этих примеров — проиллюстрировать студентам принцип структурного программирования, рассмотренный в гл. 4. При использовании языков программирования таких, как Кобол или ПЛ/1, вы должны избегать использования операторов GO TO и применять те формирующие блоки, которые рассматривались на классных занятиях. При использовании Фортрана или языка ассемблера вы должны уметь показать, что любой оператор GO TO преобразуется в программу такого вида, которую мы обсуждали и назвали «черным ящиком». Вы можете быть не согласны с принципом исключения из программы операторов GO TO, но одна из основных задач этих упражнений состоит в том, чтобы заставить вас проверить этот принцип в нетривиальном случае. Поэтому любой необязательный оператор GO TO, встретившийся в вашей программе, будет расцениваться как проявление вашей профессиональной слабости, лени или упрямства.

3. Третья цель, непосредственно вытекающая из первых двух,— получить хорошо построенную модульную программу, легкую для чтения и удобную в использовании. Постарайтесь создать такую программу, которую бы мог легко прочитать и понять ваш преподаватель или даже такую программу, которую бы вы с гордостью могли показать вашим друзьям.

4. Еще одна цель, к которой вы должны стремиться,— это написать полностью законченную программу. Но эта задача не является столь важной, как первые три. И действительно, ограничения по времени могут оказаться настолько жесткими, что вы не успеете закончить написание программы. Для вас более важным, чем написание вполне законченной, но запутанной программы, должно быть составление иерархической программы в таком виде, когда верхние

уровни завершены полностью, а нижние из-за недостатка времени подробно не расписаны. В то же время, если вы напишите только программу верхнего уровня (т. е. главную программу), никто не оценит ваших усилий, даже если эта программа будет изящно и искусно построена. Здесь необходим разумный компромисс.

5. И последняя по счету и по значению цель этих упражнений — получение эффективной программы. Однако недостаток времени не дает возможности уделить достаточное внимание получению эффективной программы, и к тому же эта задача и не ставится перед данными упражнениями. Очевидно, что вы постараетесь избежать явных оплошностей при написании программы, таких, например, как повторение вычисления 100 раз вместо использования программного цикла, повторяющего одно и то же вычисление 100 раз. Сдругой стороны, не следует пренебрегать и сложившейся практикой программирования (особенно практикой структурного программирования), от методов которой вы свободно могли бы отказаться для более эффективного решения задачи. Например, следует не ограничивать себя в использовании таких построений, при которых используются вложенные операторы IF.

В. ПРАВИЛА ИГРЫ

1. План курса предусматривает обычные лекции и семинары, чередующиеся с занятиями, посвященными решению задач. В случае удачи студенты могут уговорить преподавателя также присоединиться к ним для решения таких задач.

2. Весь класс следует разбить на группы или *бригады,* в каждую из которых должно входить от 3 до 5 человек. Каждая *бригада* должна выбрать своего *представителя* и *секретаря,* который должен регистрировать все интересные предложения и высказывания членов группы. Из-за ограниченности времени, а также из-за того, что члены бригады недостаточно знакомы друг с другом, выбирать руководителя бригады необязательно. Возможно, что все члены бригады предпочтут выступать на равных правах. После того как упражнение выполнено, будет небезынтересно еще раз обсудить вопросы, поднятые Вейнбергом в его книге «Психология программирования для ЭВМ»[1)], по коллективному и обезличенному программированию.

3. Наиболее важным для каждой группы является определение тех целей, к которым следует стремиться при решении задачи. Для удобства составы бригады могут подбираться на основе используемого языка программирования. Например, могут быть созданы бригады Фортрана, Кобола и т. д. По такому же принципу при разбиении класса на бригады можно взять за основу тип используемой

1) Weinberg, The Psychology oi Computer Programming.

ЭВМ. Например, бригада Системы IBM/370, бригада Honeywell и т. д. Очень существенным является то, чтобы каждая бригада постаралась определить, что она будет максимизировать и на каком аспекте задачи она будет концентрировать в основном свое внимание. Ниже приводятся возможные варианты такого выбора:

а) Одна из бригад может сосредоточить свое внимание на достижении максимальной *эффективности,* т. е. наилучшего использования времени ЦП, памяти и других ресурсов ЭВМ.

б) Другая бригада может сосредоточить свои усилия на получении программы, простой и удобной для сопровождения, т. е. на разработке такой программы, которая имела бы наибольшие шансы для успешной передачи отделу сопровождения.

в) Еще одна бригада могла бы поставить перед собой задачу написания и сверки (безмашинной проверки) как можно большей части программы. Ее целью должно быть написание всей программы целиком к окончанию занятия.

4. Весьма желательно иметь две или три так называемые «рабочие бригады», которые бы попытались разработать и написать программу, преследуя при этом одновременно несколько целей из числа упомянутых выше. Две или три бригады могли бы выступать в роли судей, оценивающих и критикующих деятельность остальных бригад. Судейские бригады должны испольаовать свое время для разработки перечня вопросов или критериев оценки, по которым они считают целесообразным оценивать работу рабочих бригад. Судейские бригады должны участвовать в работе рабочих бригад и прислушиваться к их замечаниям, высказываниям в процессе работы.

5. Очевидно, каждая бригада должна стремиться выполнить как можно больший объем работ, что является основной задачей секретаря бригады. При этом любые заметки, памятные или черновые записи, а также рабочие документы и другие реальные результаты их труда должны быть по возможности размножены (при наличии времени) для использования их другими учащимися класса.

Приложение А. Денежная задача

I. Исходная информация

А. ФИНАНСОВЫЕ МОДЕЛИ

В настоящее время большинству фирм стало выгодным использовать ЭВМ и выполнять на них специальные программы, помогающие их высшему руководству оценивать и принимать различные решения по финансовым вопросам, связанным с изменениями в выпуске продукции, капитальными затратами, затратами на рекламу и т. д. Такие программы называют обычно «информационными управляющими системами», а мы будем их называть «финансовыми моделями».

Некоторые исходные данные и предпосылки для применения такой модели рассматривались в разд. 2.1.2, и необходимость использования такой программы и ее эффективности нетрудно показать. В любом коммерческом предприятии руководитель имеет дело с некоторым количеством переменных параметров, которые он должен учитывать. К их числу относятся: сбытовые возможности; производственные планы; реклама и ряд других факторов, которые могуть оказать влияние на число клиентов; политика цен, которая в конце концов определяет доходы фирмы и, конечно, различные затраты, которые фирма должна производить.

Данные, получаемые на выходе таких моделей, зависят от того, что необходимо знать руководству; однако обычно всегда имеется некоторая основная информация, которой хотел бы располагать любой трезво мыслящий руководитель. Наиболее важной из такого рода информации является отчет о прибылях и убытках; в таком отчете приводятся помесячный итог о доходах, затратах, прибылях (или убытках), а также финансовый баланс (общая итоговая сумма прибылей или убытков) и прочие величины.

Б. ФИНАНСОВЫЕ МОДЕЛИ ДЛЯ ФИРМ, ПРЕДОСТАВЛЯЮЩИХ ВЫЧИСЛИТЕЛЬНЫЕ УСЛУГИ В РЕЖИМЕ РАЗДЕЛЕНИЯ ВРЕМЕНИ

Класс задач, приводимый в этом приложении, включает и задачу разработки финансовой модели для фирмы, предоставляющей вычислительные услуги в режиме разделения времени. В данном слу-

чае производимая продукция как таковая отсутствует; основной целью такой фирмы является привлечение как можно большего числа клиентов, использующих ее систему разделения времени и оплачивающих эти услуги по очень высоким ценам, которые установились почти повсеместно за последнее время. Одним из основных факторов, влияющих на успешную деятельность такого предприятия, является количество торговых агентов, рекламирующих услуги фирмы, и их способность убедительно преподносить эту рекламу. А то, скольдолгоклиентбудет пользоваться услугами такой фирмы, прежде чем его перетянет аналогичная фирма, предоставляющая большую или лучшую (или же эффективнее рекламируемую) систему, определяется техническими характеристиками последней (ее производительностью, надежностью, ассортиментом предоставляемых прикладных программ, языков программирования и т. д.). Нетрудно приблизительно оценить, какой доход может получить фирма от каждого клиента. Очевидно, что затраты фирмы складываются из расходов на вычислительное оборудование, средства связи, на содержание персонала и т. д.

Как мы увидим в следующем разделе, для функционирования финансовой модели предоставления услуг в режиме разделения времени (которую мы будем называть ДЕНЬГИ) необходимо иметь перфокарты, с которых в ЭВМ должны считываться параметры, задаваемые руководителем: число торговых агентов и т. д. Одним из важных моментов в такой задаче является форма записи переменной «DO-loop». Типичной будет ситуация, когда руководитель пожелает сделать несколько прогонов финансовой модели, меняя число торговых агентов и оставляя неизменными остальные параметры. Например, он пожелает выполнить серию прогонов модели, задавая число торговых агентов (NSALESMEN) равным 5, 10, 15, 25, 30. В этом случае он должен записать карту параметров в виде:

```
NSALESMEN = 5,30,5
```

Для программы эта запись означает, что пользователь задает начальное и конечное значение, а также приращение таким же образом, как это делается в операторе DO-loop Фортрана или PERFORM VARYING Кобола.

Важно понимать, что пользователь может пожелать изменить любой из имеющихся в программе параметров. Например, он может задать исходные данные для ввода в следующем виде:

```
NSALESMEN = 5,30,5
NPROGRAMMERS=1,10,1
```

В этом примере руководитель указывает, что он хочет получить результаты вычислений для случая 5 торговых агентов и одного программиста, после чего повторить вычисление для 10 торговых агентов и одного программиста, а затем повторять эти вычисления

до тех пор, пока число торговых агентов не станет равным 30. После этого вычисления должны быть повторены для 5 торговых агентов и 2 программистов, для 10 торговых агентов и 2 программистов и т. д. Важно иметь в виду, что в общем случае директор, используя этот режим вложенных циклов вычислений типа DO-loop, может задавать изменение всех параметров задачи. При этом общее число возможных комбинаций переменных величин (или другими словами число операций, которые будут выполняться вашей программой) будет равно *произведению* чисел значений, которые будет принимать каждая переменная величина. В связи с этим очень полезно ввести в вашу программу некоторую подпрограмму контроля ошибок, которая бы следила за тем, чтобы программист не мог запросить неоправданно большое число изменений параметров.

II. Задание на разработку программы ДЕНЬГИ

А. ОБЩИЕ СВЕДЕНИЯ

Для работы программы ДЕНЬГИ требуется ввести около 50 перфокарт с параметрами, которые задаются руководителем для описания некоторого варианта предоставления вычислительных услуг в режиме разделения времени. Для целей задания этих исходных данных параметры будут обозначаться N1, N2, N3 и т. д.; однако имеется возможность обозначать входные параметры в некоторой мнемонической форме, например NUMBEROFSALESMEN.

Считайте, что исходные данные для программы ДЕНЬГИ будут вводиться через устройство чтения перфокарт, хотя иногда руководитель может вводить эти данные непосредственно с терминала, установленного на его рабочем столе. При этом у вас имеется возможность выбрать действительный и подробный формат записи вводимых параметров.

Б. ПРЕДПОЛОЖЕНИЯ О КЛИЕНТАХ

1. Предположим, что фирма по предоставлению вычислительных услуг в режиме разделения времени начинает свою деятельность 1 числа первого месяца года (например, 1 января 1975 г.; по желанию этот день может быть любым днем года). И в этот знаменательный день N1 торговых агентов начинают рекламировать услуги, которые предоставляются фирмой.

2. После этого в первый день каждого месяца руководство фирмы будет нанимать дополнительно торговых агентов в количестве, равном некоторому числу N2, умноженному на число клиентов, пользовавшихся услугами фирмы в предыдущий месяц. Например, 1 января фирма начала свою деятельность, имея N1 торговых агентов, а 1 февраля руководство фирмы наняло дополнительное число торго-

вых агентов в количестве N1 умноженном на число клиентов, существовавших на 1 января.

3. По различным причинам каждый месяц N3 % торговых агентов покидают фирму. Однако давайте предположим, что, прежде чем торговый агент уйдет, он отработает не меньше месяца. Будем считать также, что все торговые агенты в фирме в одинаковой степени чувствуют необходимость перехода на более выгодное место, т. е. и младшие, и старшие торговые агенты в равной степени имеют желание уйти из компании.

4. Руководство установило верхний предел числа торговых агентов равным N4.

5. В течение первого месяца работы торговый агент может привлечь к использованию услуг фирмы N5 клиентов. Предположим также, что клиенты начинают пользоваться системой в первый день следующего месяца (ограничение, накладываемое в нижеследующем разделе В). Таким образом, если торговый агент начал работать на фирме 1 января и в январе он привлек N5 клиентов, то официальными клиентами фирмы они станут только 1 февраля.

6. В каждый последующий месяц своей работы торговый агент сможет привлекать на N6 клиентов *больше,* чем в предыдущем месяце. Таким образом, за второй месяц своей работы торговый агент привлечет (N5+N6) клиентов, за третий месяц — (N5+2N6) клиентов и т. д.

7. Максимально за месяц торговый агент может привлекать не более N7 клиентов.

8. В силу различных причин N8% клиентов, не удовлетворенных системой разделения времени, будут каждый месяц прекращать пользоваться ее услугами. Это распространяется в равной степени как на старых, так и на новых клиентов. Таким образом, если торговый агент привлекает 1 января X клиентов, то N8% из них оказываются разочарованными и не воспользуются услугами фирмы даже до того, как 1 февраля они станут «официальными» клиентами.

В. ПРЕДПОЛОЖЕНИЯ О ДОХОДАХ

1. Каждый клиент первые N9 месяцев, после того как он становится официальным клиентом, затрачивает на то, что преобразует свои программы, изучает технические руководства, старается установить, соответствующим ли образом установлен его терминал и знакомится с системой в принципе. Таким образом, в первые N9 месяцев он не приносит дохода.

2. В течение своего первого «рабочего» месяца клиент использует N10 «единиц» машинного времени. Одна такая единица представляет собой некоторую сумму времени ЦП, времени ввода-вывода, времени подключения терминала, а также времени использования других ресурсов системы.

3. В каждый из последующих месяцев клиент использует на N11 единиц времени больше, чем в предыдущий месяц. Таким образом, во второй рабочий месяц клиент использует (N10+N11) единиц, в третий — (N10+2N11) единиц и т.д.

4. Через N12 месяцев использования системы клиент начинает приносить максимальный доход; с этого момента он каждый месяц использует одно и то же число единиц машинного времени.

5. В первый месяц своей деятельности фирма устанавливает цену за единицу машинного времени равной N13 долл.

6. N14% всех доходов компания должна возвращать клиенту из-за низкой производительности, потери данных, аварий системы и других неисправностей.

7. Цена за предоставление услуг в режиме разделения времени ежемесячно будет изменяться на N15% с целью учета влияния местной конкуренции (или чтобы учесть ее отсутствие). При этом величина N15 может быть как положительной, так и отрицательной.

8. Счет за пользование услугами в предыдущий месяц представляется клиенту для оплаты в первый день следующего месяца. Предположим, что клиент не оплачивает счет в течение N16 месяцев после его получения.

Г. ПРЕДПОЛОЖЕНИЯ О ЗАТРАТАХ

1. В число основных затрат при обеспечении услуг в режиме разделения времени входят следующие:

а) Затраты на оборудование
- ЭВМ — основная комплектация и обычный набор внешних устройств.
- Средства связи — модемы, концентраторы, каналы связи и т. п.
- Счетно-аналитические средства подготовки — ручные перфораторы, устройства дублирования и т. п.

б) Затраты на здание и инженерные сооружения

в) Административные расходы (телефон, реклама и т. п.)

г) Оплата персонала
- Программисты
- Операторы
- Техники по обслуживанию оборудования, установленного у клиентов
- Служащие канцелярии
- Руководство
- Сбытовой персонал

2. Будем считать, что начальная заработная плата всего персонала ежегодно увеличивается на N17%. Это означает, что если в тот момент, когда фирма начинала свою деятельность, средняя заработная плата одного программиста в первый месяц составляла X долл.,

то через год средняя заработная плата в месяц одного программиста будет составлять X+N17% долл.

3. Предположим, что заработная плата каждого служащего ежегодно увеличивается на N18%. При этом повышение заработной платы служащему производится после того, как он проработал на фирме очередной год.

4. К основной заработной плате следует прибавить N19% на налоги, еграхованиеслужащих и прочиедополнительныевознаграждения.

5. Заработную плату всем служащим за предыдущий месяц фирма выплачивает первого числа следующего месяца. В это же время фирма получает счета за машинное время, аренду и административные расходы за прошедший месяц. Оплата же этих счетов фирмой производится с задержкой на один месяц.

Д. ПРЕДПОЛОЖЕНИЯ О ЗАТРАТАХ НА ОБОРУДОВАНИЕ

1. Месячная арендная плата за пользование ЭВМ составляет: N20+N21x4nano клиентов системы в прошлом месяце (долл.). Такой вид арендной платы отражает ее действительное содержание: основная арендная плата за пользование ЭВМ плюс плата за использование дополнительных пакетов дисков, разнообразных устройств ввода-вывода, плата за использованное машинное время и т. п. Вторая составляющая арендной платы прямо пропорциональна числу клиентов системы.

2. Затраты на средства связи составляют: N2+N23xЧисло клиентов в прошлом месяце (долл.).

3. Ежемесячные затраты на подготовку данных средствами счетно-аналитических устройств составляют: N24+N25xЧисло клиентов в прошлом месяце (долл.)

4. Плата за аренду зданий составляет: N26+N27xЧисло человек в штате (долл.).

5. Расходы на электричество, плата за коммунальные услуги, затраты на содержание уборщиков и сторожей составляют в месяц: N29+N29xЧисло служащих в штате (долл.).

6. Административные расходы за месяц составляют N30+N31 x x Общее число служащих в штате (долл.).

Е. ПРЕДПОЛОЖЕНИЯ О СОСТАВЕ ШТАТА

1. Число программистов в штате составляет: N31+N33 x Число клиентов в предыдущем месяце. Если в месяце N1 число клиентов оказывается меньше, чем в предыдущем месяце N, то это означает, что часть программистов следует уволить (или перевести на неполный рабочий день). При возникновении столь неблагоприятной ситуации следует считать, что все программисты в одинаковой мере

должны рассматриваться в качестве кандидатов на увольнение. Однако об этом нет смысла сообщать при приеме программиста на работу; между прочим, начальная средняя заработная плата программиста в момент, когда компания начинает свою деятельность, в первый месяц составляет N34 долл. в месяц.

2. Число операторов в штате составляет: N35+N36xЧисло клиентов в предыдущем месяце; если необходимо, некоторая часть операторов может быть уволена таким же образом, как и часть программистов. Начальная заработная плата оператора в первый месяц составляет N37 долл. в месяц.

3. Такие же предположения распространяются и на техников, обслуживающих оборудование, установленное у клиентов; их число в штате равно: N38+N39xЧисло клиентоввпредыдущем месяце. Если необходимо, часть техников следует уволить, так же как и программистов. Начальная заработная плата техников составляет N40 долл. в месяц.

4. О числе торговых агентов было сказано ранее в разд. Б. Их начальная заработная плата составляет N41 долл. в месяц.

5. Число руководителей в штате равно: N42+N43xЧисло служащих, указанных в п. 1—4. Начальная заработная плата руководителя в первый месяц его работы составляет N44 долл. в расчете за месяц.

6. Число сотрудников канцелярии в штате равно: N45+N46 x x Общее число служащих, перечисленных в п. 1—5. Начальная заработная плата сотрудников канцелярии составляет N 47 долл. в месяц.

Ж. ПРОЧИЕ ПРЕДПОЛОЖЕНИЯ

1. Величина N48 представляет собой число месяцев, в течение которых будет использоваться данная модель. То есть руководитель может задавать период времени, в течение которого он сможет смоделировать показатели прибылей и убытков — 12, 36 и 109 мес.

2. Величина N49 представляет собой максимальное число клиентов, которых может обслужить система. Это ограничение возникает вследствие технических характеристик самой системы, т. е. ограничений, связанных с числом линий, модемов и т. д.

Приложение Б. Задача фирмы "Независимый удобритель"

КОМУ: Нашему новому суперпрограммисту
ОТ КОГО: От президента фирмы «Независимый удобритель»
ДАТА: 15 марта 1974 г.
КРАТКОЕ СОДЕРЖАНИЕ: О разработке информационной системы для руководства

I. Введение

1. Уважаемый мистер X! По рекомендации анонимного, но печально известного в международных кругах американского консультанта по обработке данных на ЭВМ я имею честь пригласить Вас на службу в отдел обработки данных на ЭВМ фирмы «Независимый удобритель» со скромной заработной платой 50 тыс. долл. Мы уверены, что Вы обладаете способностью программировать со скоростью, превышающей скорость полета пули, обходя такие препятствия, как высокие горы бумажных распечаток, и поражая всех своей виртуозностью. В качестве Вашей первой задачи я хотел бы предложить Вам разработать несложную информационную систему для руководства, которая бы помогала мне как президенту фирмы принимать более точные решения по вопросам текущей деятельности нашей организации.

2. Эта система — в действительности всего лишь одна программа — должна иметь дело с информацией, которая находится в нашем файле открытых заказов. Файл записан на обычной магнитной ленте и содержит информацию о заказах, поступивших от клиентов, но еще не обслуженных. Новые записи вносятся в файл (в процессе прогона программы обновления файла) по мере поступления заказов от клиентов. Впоследствии может оказаться, что некоторые записи требуют изменения из-за того, что в них имеются ошибки, или из-за того, что заказ клиента изменился (например, клиент аннулировал заказ до его выполнения). В большинстве случаев записи удаляются из файла, когда заказанная продукция отправлена клиенту. Более подробные сведения о структуре файла содержатся в разд. II.

II. Структура файла открытых заказов

1. Файл открытых заказов записан на магнитной ленте и может занимать одну или более бобин. Некоторые физические характеристики файла (такие, как плотность записи, разбиение информации на блоки, информация, используемая для разметки ленты) не

представляют для вас интереса. Вы можете считать, что один из рядовых системных программистов, входящих в наш штат, напишет для вас три подпрограммы, которые можно ВЫПОЛНЯТЬ или ВЫЗЫВАТЬ. Вот эти подпрограммы:

XOPEN должна вызываться перед считыванием любой записи. Эта подпрограмма будет выполнять все необходимые подготовительные функции, такие, как контроль метки ленты, подгонка ленты к началу файла и т. д.

XREAD будет считывать из файла одну логическую запись. Вы совместно с системным программистом должны решить, каким образом в этой подпрограмме указывать область, куда должна считываться запись.

XCLOSE должна вызываться каждый раз, когда вы заканчиваете использование файла. Она выполняет все необходимые действия по закрытию файла.

Каждая из этих подпрограмм будет устанавливать некоторый флажок перед возвратом управления вызывающей программе. Этот флажок, называемый XRESULT, будет иметь следующие значения:

XRESULT = 0 запрашиваемая операция выполнена успешно.
1 при выполнении запрашиваемой операции возникла неисправимая ошибка; например, ошибка по четности, отказ накопителя на магнитной ленте и т. п.
2 была обнаружена метка конца файла.

2. Файл открытых заказов содержит три различных типа записей: записи о клиентах, записи о поступивших заказах и записи спецификаций заказов. При необходимости к записи добавляется соответствующее число заполнителей или пустых символов с тем, чтобы обеспечить одинаковую длину всех записей. Запись о клиенте должна рассматриваться в качестве главной записи. Каждой записи о клиенте может соответствовать одна или несколько записей о поступивших заказах. В то же время каждой записи о поступившем заказе может в свою очередь соответствовать несколько записей спецификации заказа. Если какой-либо клиент сделал один или несколько заказов и они еще не выполнены, в файле на него заводится только одна запись о клиенте. В этой записи указаны номер счета клиента, адрес и другая информация общего характера. Для каждого отдельного заказа, поступившего от клиента, будет формироваться единственная запись о полученном заказе, размещенная в файле непосредственно *за записью* о клиенте. Запись о поступившем заказе указывает дату принятия заказа, торгового агента, принявшего заказ, адрес отправки и другую информацию, имеющую отношение к этому заказу. С каждым поступившим заказом связаны

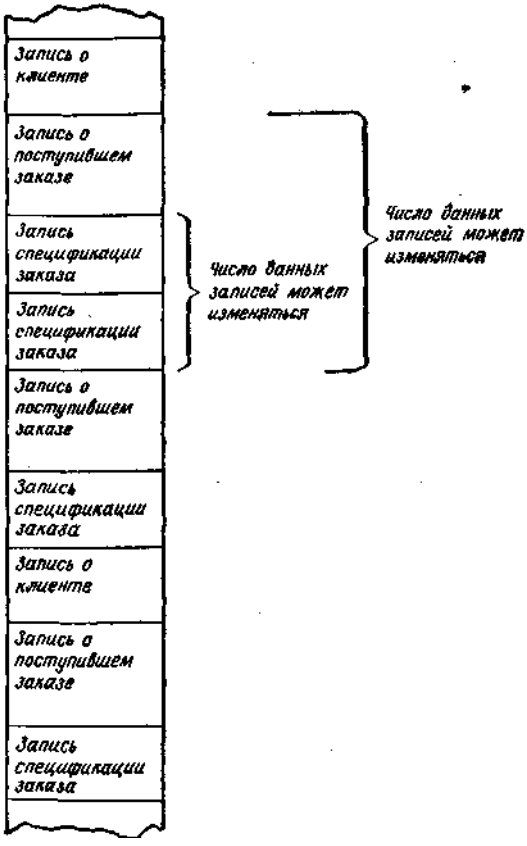


Рис. П.Б.1. Структура файла открытых заказов.

одна или более записей спецификации заказов — например, вслед за записью о поступившем заказе может следовать три записи спецификации заказа, указывающие на то, что он требует поставки определенного количества продукции наименования X, определенного количества продукции наименования Y и определенного количества продукции наименования Z. Назначение записи спецификации заказа — указывать тип продукции, объем продукции, подлежащей поставке, действующую цену в день продажи и другую информацию подобного рода.

3.. Заметим, что за записью о клиенте может следовать различное число записей о поступивших заказах, но что это число не может быть меньше 1. Таким же образом вслед за записью о поступившем заказе должна следовать одна или более записей спецификации заказа. Структура файла в общих чертах показана на рис. П.Б.1.

КОД ТИПА ЗАПИСИ	1	СИМВОЛ
НОМЕР СЧЕТА	6	
ИМЯ КЛИЕНТА	30	
1-я СТРОКА АДРЕСА КЛИЕНТА	20	
2-я СТРОКА АДРЕСА КЛИЕНТА	20	
3-я СТРОКА АДРЕСА КЛИЕНТА	20	
ГОРОД	20	
ШТАТ	20	
НОМЕР ТЕЛЕФОНА	10	
НОМЕР ТЕЛЕКСА	20	
	167	СИМВОЛОВ

Примечание: КОД ТИПА ЗАПИСИ для данного типа записи имеет значение I,

Рис. П. Б.2. Формат записи о клиенте.

КОД ТИПА ЗАПИСИ	I	СИМВОЛ
НОМЕР СЧЕТА	6	
НОМЕР ПОСТУПИВШЕГО ЗАКАЗА	8	
ИДЕНТИФИКАЦИОННЫЙ КОД ТОРГОВОГО АГЕНТА	3	
ДАТА ЗАКАЗА — ГГММДД	6	
ИНФОРМАЦИЯ О ДОСТАВКЕ:		
ИМЯ КЛИЕНТА	30	
1-я СТРОКА АДРЕСА ДОСТАВКИ	20	
2-я СТРОКА АДРЕСА ДОСТАВКИ	20	
3-я СТРОКА АДРЕСА ДОСТАВКИ	20	
ГОРОД	20	
ШТАТ	20	
КОД РАЙОНА, ГДЕ БЫЛ ПРИНЯТ ЗАКАЗ	1	
ОБЩАЯ СУММА ЗАКАЗА В ДОЛЛ.	6	
ЗАПОЛНИТЕЛИ (ПРОБЕЛЫ)	6	
	167	СИМВОЛОВ

Примечание: КОД ТИПА ЗАПИСИ для данноготипазапиеиимеетзначение2.

Рис. П. Б.3. Формат записи о поступившем заказе.

4. Структуры записей показаны: о клиентах — на рис. П.Б.2, о поступивших заказах — на рис. П.Б.3, о спецификации заказа — на рис. П.Б.4. Следует отметить, что все записи, составляющие файл, имеют одинаковую длину. Кроме того, каждая запись по смысловому содержанию является самоопределенной: первое поле записи указывает на тип записи — запись о клиенте, запись о поступившем заказе или запись спецификации заказа.

5. Основное назначение вашей программы — выдача на печать записей, выбираемых по некоторому признаку из файла открытых заказов, пользуясь которыми я мог бы получить информацию о имеющихся заказах на продукцию определенного вида (например,

КОД ТИПА ЗАПИСИ	1	СИМВОЛ
НОМЕР СЧЕТА	6	
НОМЕР ПОСТУПИВШЕГО ЗАКАЗА	8	
НОМЕР СПЕЦИФИКАЦИИ	Ю	
НОМЕР ПРОДУКЦИИ	4	
ИДЕНТИФИКАЦИОННЫЙ КОД ТОРГОВОГО АГЕНТА	3	
ДАТА ПРИНЯТИЯ ЗАКАЗА-ГГММДД	6	
ДАТА ОТПРАВКИ ЗАКАЗА ПО ПЛАНУ — ГГММДД	6	
ДАТА ОТПРАВКИ ПОСЛЕДНЕЙ ПАРТИИ — ГГММДД	6	
КОЛИЧЕСТВО	6	
ЦЕНА ЕДИНИЦЫ ПРОДУКЦИИ В ДОЛЛ.	6	
ОБЩАЯ СТОИМОСТЬ В ДОЛЛ,	6	
ПРОЦЕНТ СКИДКИ	2	
ЧИСТАЯ ЦЕНА	6	
ЗАПОЛНИТЕЛИ (ПРОБЕЛЫ)	91	
	167	СИМВОЛОВ

Примечание: КОД ТИПА ЗАПИСИ для данного типа записи имеет значение 3.

Рис. П.Б.4. Формат записи спецификации заказа.

перепревшее гуано), о заказах, ждущих выполнения и принятых каким-либо одним торговым агентом и т. д. Исходная информация будет задаваться вашей программе в виде нескольких отдельных групп управляющих карт, т. е. я могу затребовать, чтобы в результате однократного прогона вашей программы была выдана справка о всех ожидающих выполнения заказах на красное гуано, а после этого — справка о всех ожидающих выполнения заказах клиента 1234.

Такой подход к решению нашей задачи может показаться не очень удобным, но он дает нам возможность сформировать несколько различных справок за один проход через наш весьма большой файл открытых заказов. Формат входных карт описан более подробно в разд. Ш.

6. Для меня не имеет существенного значения, в каком формате будут выдаваться справки. Я только полагаю, что каждая справка должна иметь несложный заголовок, где необходимо указывать признак, по которому производился выбор записей из файла (т. е. тот признак, который задавался в управляющих картах). Кроме того, я полагаю, что записи, считанные с магнитной ленты, вы должны выводить на печатающее устройство в таком формате, который удобен для чтения. Заранее хочу поставить вас в известность о том, что в части формата справок у меня есть некоторые соображения и поэтому может оказаться, что впоследствии мне придется просить вас ввести небольшие изменения в этой части программы.

III. Формат входных данных для программы

1. Ваша программа будет решать, какие записи должны быть выбраны из файла открытых заказов, на основании информации, получаемой при вводе управляющих карт. Для простоты будем считать, что входная информация вводится не обязательно с перфокарт, а лишь имеет формат, принятый нами для управляющих карт. Ввод всех ваших управляющих карт будет организован с помощью трех подпрограмм, которые будут написаны для вас одним из рядовых системных программистов нашей фирмы. Вот перечень этих подпрограмм:

YOPEN открывает входной файл управляющей карты. Эта подпрограмма выполняет все необходимые подготовительные операции, контроль меток и т. д.

YREAD эта программа будет считывать в память информацию в формате 80-колонной перфокарты. Вы можете сформулировать для системного программиста точные правила вызова этой подпрограммы таким образом, что сможете точно определять, куда будет помещаться содержимое карты.

YCLOSE закрывает входной файл управляющей карты.

Все три подпрограммы перед возвратом управления вызвавшей их программе будут устанавливать некоторый флажок. Этот флажок, называемый YRESULT, будет иметь следующие значения:

YRESULT =0 запрошенная операция выполнена успешно.
1 при выполнении запрошенной операции произошла неисправимая ошибка (ошибка четности).
2 была обнаружена метка конца файла.

2. Каждая *группа* управляющих карт (числокартвтакойгруппе может быть различным для различных прогонов программы) будет содержать неодинаковое число карт, каждая из которых будет задавать определенное условие, без соблюдения которого запись нельзя будет выбрать из файла открытых заказов для выдачи на печать. Например, вполне возможно, что первая группа управляющих карт будет содержать пять отдельных карт в следующем общем виде:

A
B
C
D
E

Это означает, что, прежде чем запись сможет быть выбрана и отпечатана, необходимо, чтобы были выполнены условия A и B и C и D и E (одновременно).

3. Как уже было сказано, каждым прогоном программы можно управлять несколькими группами управляющих карт; при этом каждый такой «пакет» управляющих карт выделяется с помощью одной карты, на которой в первых четырех символьных позициях отперфорировано слово STOP. Таким образом, возможен случай, когда ваша программа будет считывать группу управляющих карт в виде:

```
AI
B1
C1
STOP
A2
B2
STOP
A3
B3
```

Каждая запись, которая будет считана из файла, должна быть отнесена к своей группе управляющих карт. Например, предположим, что первая запись из файла удовлетворяет условиям A1, B1 и C1; тогда эта запись должна быть выбрана и отпечатана в первой выходной справке. Предположим, что первая запись не удовлетворяет условиям A2 и B2; тогда она не должна выбираться и выводиться при выдаче второй справки. Предположим также, что эта запись удовлетворяет условиям A3 и B3; тогда та же самая первая запись должна быть выбрана и выведена на печать при выдаче третьей справки и т. д. Следовательно, становится понятным, что одна и та же запись может выдаваться на печать при выдаче нескольких различных справок.

4. Каждая управляющая карта состоит из двух полей: кода заявки и поля параметров. Код заявки занимает самые левые символьные позиции управляющей карты и может иметь длину до 10 символов. Поле параметров всегда будет начинаться с символьной позиции 11; обычно это поле должно состоять из одного или двух целых чисел. Для примера ниже приводится видтипичной управляющей карты:

```
CUSTOMER      123456
```

Всего существует 8 типов управляющих карт; ниже подробно описан каждый из этих типов. Пакет управляющих карт может со-

стоять из любой комбинации управляющих карт, которые могут располагаться в нем в произвольном порядке. Однако при этом управляющая карта некоторого типа может входить в пакет не более одного раза. Следовательно, управляющая карта КЛИЕНТ (CUSTOMER) может входить в один пакет управляющих карт не более одного раза.

5. Для карты КЛИЕНТ требуется поле параметров, состоящее из одного 6-значного целого числа. Это число указывает 6-значный номер счета клиента и означает, что все записи клиентов, все записи поступивших заказов и все записи спецификаций заказов, содержащие этот номер, должны быть включены в справку, которая выдается по запросу данного пакета управляющих карт. Следует отметить, что для упрощения процесса обработки управляющей карты и записи клиентов, и записи поступивших заказов, и записи спецификаций заказов, входящие в файл, содержат в своем составе номер счета клиента. Если в управляющей карте отсутствует поле параметра или оно превышает отведенную для него длину, или имеет неправильный формат, другими словами, в нем не записана 6-значная цифра, начинающаяся с 11 позиции, ваша программа должна выдавать на печать сообщение об ошибке.

6. Для карты НОМЕР ПОСТУП ЗАКАЗА (PONUMBER) требуется поле параметра, состоящее из одного 8-значного целого числа. Это означает, что в выходную справку должны быть включены все записи поступивших заказов и записи спецификаций заказов, содержащие номер поступившего заказа, указанный в этом поле. Необходимо заметить, что при воздействии этой управляющей карты запись клиента на печать выводиться не должна. Если в этой карте отсутствует поле параметра, или оно превышает отведенную для него длину или имеет неправильный формат, должно печататься сообщение об ошибке.

7. Для управляющей карты СПЕЦИФИКАЦИЯ (ITEM) требуется поле параметра, состоящее из одного 10-значного целого числа. Оно указывает только одну запись спецификации заказа, которая содержит именно это число и должна быть включена в выходную справку. Следует отметить, что каждая запись спецификации, входящая в файл открытых заказов, содержит в этом поле свой особый номер и, следовательно, к каждой спецификации, выданной клиентом, можно обращаться по отдельности с помощью программы выборки, подобной той, что описывается здесь. Если в этой карте отсутствует поле параметра или оно превышает отведенную для него длину, или имеет неправильный формат, то должно печататься сообщение об ошибке.

8. Для управляющей карты ПРОДАНО ТОРГ АГЕНТОМ (SOLDBY) требуется поле параметра, состоящее из одного 3-значного целого числа. Эта карта указывает, что в выходную справку должны быть включены все записи поступивших заказов и записи

спецификаций, содержащие идентификационный номер торгового агента, который записан в поле параметра этой карты. Следует отметить, что при начальном открытии заказа клиентом запись поступившего заказа и все связанные с ней записи спецификаций содержат один и тот же номер торгового агента. Однако в результате последующих изменений файла могут произойти отдельные модификации записей спецификаций и (или) записей поступивших заказов, а также исключение некоторых записей и добавление новых в соответствии с указаниями торговых агентов, которые принимают такие запросы от клиентов, чьи заказы уже приняты, но еще не выполнены. Поэтому вполне возможно, что записи поступивших заказов и связанные с ними записи спецификаций будут содержать различные идентификационные номера торговых агентов. Для простоты мы можем предположить, что запись поступившего заказа будет содержать номер того торгового агента, который принял от клиента первоначальный заказ, а записи спецификаций — номер торгового агента, который выполнял какие-либо операции, связанные с этими спецификациями последним. Мы хотим, чтобы при обработке управляющей карты ПРОДАНО ТОРГ АГЕНТОМ в справку включались только те записи поступивших заказов и только те записи спецификаций, которые содержат заданный идентификационный номер торгового агента. Если в этой карте отсутствует поле параметра, содержится несколько полей параметра или имеет место неправильный формат этого поля, то должно печататься сообщение об ошибке.

9. Для управляющей карты ПРОДУКЦИЯ (PRODUCT) требуется поле параметра для записи одного 4-значного целого числа. Эта карта указывает, что все записи спецификаций, содержащие указанную в этой карте продукцию (например, гуано с запахом клубники), должны быть включены в выходную справку. Если в этой карте отсутствует поле параметра, содержится несколько полей параметра или имеет место неправильный формат этого поля, то должно печататься сообщение об ошибке.

10. Для управляющей карты ДАТА (DATA) требуется поле параметра для записи двух 6-значных целых чисел. Каждое из этих чисел имеет вид ГГММДД (год, месяц, день). Например, дата 13 ноября 1973 г. записывается как 731113. Первое из двух чисел, записываемое в поле параметра, указывает *начальную* дату; после нее ставится запятая, а за ней записывается второе 6-значное целое число — *конечная* дата. Управляющая карта ДАТА указывает, что должны быть выбраны все записи поступивших заказов и связанные с ними записи спецификаций, которые были открыты в день, приведенный в данной управляющей карте в качестве начальной даты, и во все последующие дни, но не позже дня, указанного в карте в качестве конечной даты. Таким образом, управляющая карта

указывает, что мы должны выбрать все записи, относящиеся к заказам (и связанным с ними спецификациям), открытым между 4 января и 13 ноября (включительно) 1973 г. Если в этой карте отсутствует поле параметра, содержится только одно поле параметра, число полей параметра превышает 2, даты расставлены не в надлежащем порядке или имеет место неправильный формат этого поля, то должно печататься сообщение об ошибке.

11. Для управляющей карты РАЙОН (REGION) требуется поле параметра для записи переменного количества однозначных целых чисел. Однозначные целочисленные коды обозначают различные районы сбыта. Например, число 1 может обозначать Новый Южный Уэльс. После каждого кода района кроме последнего ставится запятая, а после последнего — специальный кодО, являющийся указателем конца поля. Таким образом, типичная управляющая карта РАЙОН имеет вид

РАЙОН 1,3,2,4,0

Назначение управляющей карты РАЙОН — задавать выборку тех записей поступивших заказов и связанных с ними записей спецификаций, по которым отгрузка продукции клиентам будет производиться в *какой-либо* из указанных районов. В приведенном только что примере в выходную справку из- файла открытых заказов должны быть включены те записи, в которых указан район отправки 1 *или* 3 *или* 2 *или* 4 *и* которые, кроме того, удовлетворяют условиям других управляющих карт, входящих в тот же самый пакет управляющих карт. Максимально может указываться до 9 кодов районов сбыта, и располагаться они могут в произвольном порядке. Если в этой карте отсутствует поле параметра, код одного и того же района повторяется более одного раза, задано более 9 районов, имеет место неправильный формат этого поля или список кодов не завершается нулем, то должно печататься сообщение об ошибке.

12. Для управляющей карты КОЛИЧЕСТВО (AMOUNT) требуется поле параметра для записи одного 6-значного целого числа. Карта КОЛИЧЕСТВО указывает, что должны быть выбраны все записи поступивших заказов и связанные с ними записи спецификаций, в которых общее количество указанной в них прдукции превышает количество, заданное в этой карте 6-значным целым числом. При этом следует обратить внимание на то, что количество заказанной продукции записано в самой записи поступившего заказа и является суммой значений, внесенных в поля «количество» тех записей спецификаций, которые связаны с данной записью поступившего заказа. Если в этой карте отсутствует поле параметра, указано более одного параметра или имеет место неправильный формат этого поля, то должно печататься сообщение об ошибке.

13. Как уже было сказано, в конце каждого пакета управляющих карт должна помещаться одна карта, на которой в символьных по-

зициях 1-4 содержится слово СТОП (STOP). Последний пакет карт также оканчивается картой СТОП, но сразу за ней помещается карта, в символьных позициях 1-3 которой содежится слово КОНЕЦ (END). Таким образом, формат входной колоды имеет вид:

```
A1
B1
C1


STOP
A2
B2


STOP


An
Bn
Cn
STOP
END
```

Вы можете считать, то при любом прогоне вашей программы число пакетов управляющих карт не превышает 10.

IV. Дополнительные требования к формату

1. Кроме частных ошибок, связанных с отдельными управляющими картами, существует и некоторое число ошибок более общего характера, за которыми должна следить ваша программа. Ниже приводится перечень этих ошибок:

а) Управляющие карты отсутствуют, или, другими словами, в тот момент, когда ваша программа запрашивает выполнение операции ввода, сразу же принимается указатель конца файла.

б) Отсутствует карта КОНЕЦ.

в) Последняя карта СТОП отсутствует, а карта КОНЕЦ представлена.

г) Карта КОНЕЦ не является последней в колоде. Если входная колода составлена правильно, то при попытке считать очередную карту пссле карты КОНЕЦ должно возникать условие конца файла.

д) Входная колода состоит более чем из 10 пакетов управляющих карт.

2. Кроме уже перечисленных ошибок общего характера также могут возникнуть ошибки нескольких типов, связанные с отдельными пакетами карт. Вот перечень таких ошибок:

а) Наличие пустого пакета — две карты СТОП следуют одна за другой.

б) В пакет входит более чем одна карта с указанием типа, т. е. карта КЛИЕНТ (CUSTOMER).

в) Неправильный код заявки — код заявки в колонках 1—1б не соответствует ни одному из 8 типов, описанных ранее. Следует заметить, что такая ошибка может возникнуть из-за ошибки при перфорации КОНЕЦ или СТОП, что, кроме того, может привести к ошибкам и другого рода.

г) Неправильный формат поля параметров во входной карте некоторого типа — фактически эта ошибка относится*к ошибкам частного характера, которые уже были описаны ранее.

V. Данные на выходе программы

1. Как уже видно из приведенной информации, программа может вызываться для того, чтобы выдать на печать до 10 выходных справок. Для простоты мы будем считать, что эти выходные справки будут записываться сначала в отдельные файлы, размещенные на магнитных лентах или дисках, откуда впоследствии они будут выдаваться на печать (с помощью другой программы). Вывод справок в какой-либо из выходных файлов будет выполняться с помощью еще одного набора подпрограмм, которые также будут написаны для вас одним из рядовых системных программистов нашей фирмы.

ZOPEN открывает i-й файл
ZWRITE выводит одну логическую запись в i-й выходной файл
ZCLOSE закрывает i-й выходной файл

Системный программист, которому предстоит писать эти подпрограммы, будет согласен на любой приемлемый интерфейс с подпрограммой для указания значения i и точного задания местонахождения выходной записи.

Для каждой из этих подпрограмм требуется задание аргумента i, который указывает, какой из выходных файлов (их число может быть переменным, но не должно превышать 10) будет использоваться. Как и в случае с ранее описанными подпрограммами ввода-вывода, каждая из только что перечисленных подпрограмм перед возвратом управления исходной программе будет формировать признак результата ZRESULT, который может принимать следующие значения:

ZRESULT = 0 запрошенная операция окойчилась успешно
1 произошла неисправимая ошибка

2 превышен объем области памяти, отведенной для вывода (эта ситуация будет возникать достаточно редко и только в тех случаях, когда вывод производится на дисковое ЗУ).

2. Формат выходной справки не оговаривается, и вы можете выдавать на печать все, что сочтете необходимым. Справка должна содержать заголовок, указывающий критерий, который использовался при выборе записей. Вслед за ним должны следовать записи, выбранные из файла открытых заказов фактически в том же самом формате, в котором они записаны на ленте.

3. Наряду с обычными выходными справками ваша программа должна обеспечивать выдачу справки обо всех ошибках, которые возникли при выполнении программы. Программа должна проверять каждую управляющую карту перед началом ее обработки. Если в какой-либо из управляющих карт обнаруживается ошибка, должно печататься соответствующее сообщение об ошибке в файле справок об ошибках и данный пакет управляющих карт должен быть исключен из обработки (однако для удобства работы пользователя перед тем, как пакет будет отвергнут, было бы полезно проверить все управляющие карты из этого пакета и отпечатать все обнаруженные сообщения об ошибках). Если при выполнении программы возникает ошибка, которая не будет исправлена автоматически, в файле справок об ошибках должно печататься соответствующее сообщение об ошибке, после чего оператор должен выполнить необходимые действия для устранения ошибки (программа должна определить, какие действия необходимо выполнить оператору).

4. Вы можете считать, что файл справок об ошибках является одиннадцатым выходным файлом. Таким образом, вывод выходных результатов в этот файл может выполняться посредством вызова подпрограмм ZOPEN, ZWRITE и ZCLOSE с заданием параметра i=11.

Приложение В. Задача модификации главного файла

КОМУ: Суперпрограммисту Горацио
ОТ КОГО: От владельца фирмы
ДАТА: 19 августа 1974 г.
СОДЕРЖАНИЕ: Задание на программирование

I. Введение и общее описание

Ваше задание состоит в том, чтобы написать программу индивидуальной модификации отдельных полей в задаваемых записях нашего главного файла клиентов. Этот файл является последовательно организованным файлом и размещается на нескольких бобинах магнитной ленты. Файл построен в порядке возрастания номеров счетов клиентов.

Модификации главного файла должны выполняться вашей программой на основе данных, вводимых с перфокарт (или каким-либо другим способом, но в формате карт), представляемых в свободном формате. В каждой карте будет задаваться номер счета того клиента, чья запись должна быть модифицирована; вслед за номером счета будут задаваться одно или несколько полей, подлежащих модификации. В силу сложившегося порядка группа подготовки пользовательских данных выдает нам карты модификации, уже упорядоченные по номерам счетов клиентов.

Ваша программа должна считывать данные с этих карт, выполнять определенные проверки на правильность и затем в соответствии с заданными указаниями производить модификацию главногофайла. Кроме того, ваша программа должна выдавать на печать краткую справку о любых ошибках, возникших при ее выполнении.

II. Структура главного файла

Как уже упоминалось главный файл является последовательно организованным, размещен на магнитной ленте и упорядочен по номерам счетов клиентов. На каждого клиента в главном файле имеется одна запись; каждая запись состоит из 142 символов. Структура записи показана в табл. П.В.1.

Для большей универсальности ваша программа не должна иметь дело с такими деталями, как буферирование, группирование, ленточные метки и прочие, связанными с главным файлом. Такого рода функции будут выполняться подпрограммами, которые ваша программа может *вызывать* или *выполнять*.

Таблица П.В.1

СТРУКТУРА ГЛАВНОГО ФАЙЛА

Поле	Тип информации	Длина поля	Алфавитная или цифровая информация	Изменяется программой модификации
1	Номер счета	5	Цифровая	Нет
2	Имя клиента	30	Алфавитно-цифровая	Да
3	Адрес клиента—улица	30	То же	Да
4	Город	20	» »	Да
5	Штат (сокращенно)	2	Алфавитная	Да
6	Код	5	Цифровая	Да
7	Номер телефона (с кодом области)	10	То же	Да
8	Статус: активный или неактивный	1	Алфавитная	Нет
9	Торговый агент, ведущий этот счет	5	Цифровая	Да
10	Дата последней заявки	6	То же	Нет
11	Дата последнего платежа	6	» »	Нет
12	Текущий баланс	8	» »	Нет
13	Общая сумма сделок за текущий год	8	» »	Нет
14	Предельная сумма кредита	6	» >	Нет

Каждая запись содержит 14 полей, общая длина которых 142 символа. Назначение каждого поля указано в таблице.

Далее приводится описание этих подпрограмм:

MFOPEN открывает *входной главный файл* и *выходной главный файл;* выполняет контроль меток, инициализацию буферов и т. д.

MFCLOSE закрывает входной и выходной главные файлы

MFREAD считывает одну запись из входного главного файла в область, задаваемую программистом

MFWRITE записывает одну запись в выходной главный файл из области, задаваемой программистом

Обе подпрограммы и MFREAD и MFWRITE будут формировать переменную MFAAG, которая может принимать следующие значения:

0 чтение/запись закончены успешно.

1 возникла ошибка четности, которая не исправляется автоматически.

2 обнаружена метка конца файла (EOF) (имеет место только при выполнении подпрограммы MFREAD).

III. Структура данных, вводимых с перфокарт

Как уже упоминалось, модификация главного файла будет выполняться на основе данных, вводимых с перфокарт. Ваша программа может йолучать доступ к этим перфокартам путем вызова (или выполнения) следующих подпрограмм:

CROPEN	открывает входной файл устройства чтения
CRCLOSE	закрывает входной файл устройства чтения
CREAD	считывает содержание одной 80-колонной карты в область, задаваемую программистом

Кроме того, подпрограмма CREAD будет формировать переменную, которая может принимать следующие значения:

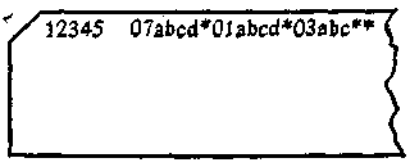
CRFLAG =	0	операция закончена успешно.
	1	возникла ошибка ввода-вывода, которая не исправляется автоматически
	2	обнаружена метка конца файла (EOF)

Данные на карте имеют следующий формат. Колонки 1—5 указывают номер счета записи, подлежащей модификации. Колонки 6—9 оставляются незаполненными и должны игнорироваться программой. Начиная с колонки 10, записывается переменное число полей в форме:

xxabcd...xyz*xxabc...xyz* и т. д.

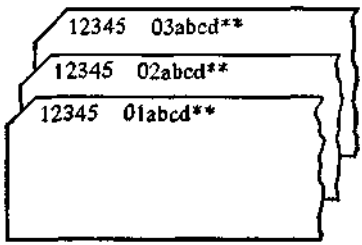
где xx представляет собой 2-значное целое число, указывающее, какое из 14 полей записи должно быть смодифицировано, а abc. . .xyz— последовательность символов переменной длины, которая должна заменить соответствующее поле в главной записи.

Каждое поле оканчивается знаком звездочка (*). Длина поля ограничивается сверху границей перфокарты. Таким образом, поле не может переходить с одной карты на другую. Последнее поле на карте обозначается двумя звездочками в конце.



Кроме того, следует добавить, что может иметь место ситуация, когда несколько карт задают модификацию одной и той же главной записи. В этом случае допустимой является следующая последова-

тельность!



IV. Требования к формату

В процессе обработки ваша программа должна следить за возникновением ошибок явного вида. При обнаружении ошибки должно печататься соответствующее сообщение об ошибке; формат этих сообщений будете определять вы. Сообщение об ошибках должно выводиться на высокоскоростное печатающее устройство с помощью следующих подпрограмм:

PROPEN	открывает файл печати
PRCLOSE	закрывает файл печати
PRWRITE	производит вывод сообщения в файле печати на устройство печати из области, задаваемой программистом.

Ваша программадолжна обнаруживать следующие типы ошибок:

1. *Недействительные номера счетов.* Если номер счета, представленный на карте, отсутствует в главном файле или записан с отступлениями от правил, вся карта должна отвергаться, а на печать должно выдаваться соответствующее сообщение.

2. Недействительные номера полей. Номер поля должен находиться в пределах от 1 до 14. Если это не так, то данное поле должноигнорироваться, асообщениеобэтомвыдаваться на печать. Последующие поля должны обрабатываться при условии, конечно, что они отвечают правилам.

3. *Недействительный вид данных.* В табл. П.В.1 показано, что различные поля должны заполняться символами вполне определенного вида — цифровыми, алфавитными или алфавитно-цифровыми. Ваша программа должна проверять это и отвергать те поля, которые заполнены знаками не того вида.

4. *Изменение полей, не подлежащих изменению.* В табл. П.В.1 показано, что пользователю рассматриваемой программы разрешено изменять не все поля; например, пользователю не разрешается изменять номер счета. Если все же пользователь попытается смоди-

фицировать какое-либо из этих полей, такая попытка должна быть отвергнута.

5. *Неправильная длина поля.* Поле данных на карте может быть короче, длиннее или одинаковой длины с соответствующим полем в главном файле. Если поле на карте длиннее, должно формироваться сообщение об ошибке, а само поле или должно игнорироваться, если в нем содержится цифровая информация, или укорачиваться, если в нем содержится алфавитно-цифровая или алфавитная информация.

Приложение Г. Программа для игры в тик-так

1. Ваше задание написать программу ЭВМ для игры в тик-так с партнером (скорее всего, человеком). Мы считаем, что правила игры вам известны. Если вы с ними не знакомы, то можете их узнать, расспросив любого студента в группе.

2. Для определенности вы можете считать, что ваша программа будет играть с партнером, который будет связан с ЭВМ через терминал, работающий в режиме разделения времени. Если вы знакомы с системами разделения времени, то можете выбрать любой тип интерфейса с пользователем терминала. Если же опыта в использовании систем разделения времени у вас нет, то для связи с пользователем вы можете воспользоваться следующими двумя подпрограммами:

```
INPUT(CHAR)
OUTPUT(CHAR)
```

Эти две подпрограммы по вашему желанию могут выполняться или вызываться. Имеется в виду, что CHAR — это поле переменной длины, состоящее из одного символа, который выводится на терминал или принимается от него.

3. Это все, что касается данного задания.

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ

Алгол, исключение GOTO 35
— ошибки программирования **341**—345
— пакеты отладочных программ 161
— реализация CASE 181
— символьные параметры 144—145
— средства реализации структурного программирования 180—181
Алгоритмы поиска 14, 159—160, 235—237
Аппаратные средства, нестандартные возможности 22
———ошибки 331—332, 341—344
Апробирование 287
Аттестация 287
Ашкрофта — Манны метод 192

Библиотека программной разработки 108
————проекта IBM 106—110
Блок-схемы, документирование проекта 74—75
— сопоставление с таблицами решений 127—129
Блочная организация программы 32
— структура в Алголе и ПЛ/1 201—202
———отсутствие в Коболе 203
————в Фортране 35, 204

Верификация 287
— дисциплина сопровождения 38
— сопоставление с тестированием 21
Восходящее проектирование 100—102
— тестирование 78—79, 300—302
———применения 93—95
———сопоставление с нисходящим 78—79
BASIC 203
BLISS **169,** 206, **213**

Гибкость программы 39—40
———методы обеспечения 40
Глобальные параметры (см. Символьные параметры)
GOTO вычисляемый 233
— исключение в структурном программировании 170—176
— использование в Фортране 204—205
— мера квалификации программиста 167
~ назначенный 231—232
—• небрежное использование 32
— ограничения 184
— частота употребления 32

Дампы (см. Отладки методы)
ДДТ реализация версии 361—375
— типы пакетов 350—352
— характеристики пакетов 352—353
Динамическое распределение памяти 8, 162

Защита от ошибок 256
————вычисляемые GOTO 279—280
————диапазоны изменения и типы данных 268—270
————индексация массивов 279, 344
————комментарии 272—273
————контролируемые действия 260—267
————контроль ввода-вывода 270—272
————множественные признаки 273—277, 343—344
————открытое объединение модулей 278
— — — сопоставление с тестированием и отладкой 256—257, 309—310
————условия передачи управления 277-278

———— цели 2Ы>
————числовые данные 267—268
————эффективность программы 258-259

Интерфейсы межмодульные 62—65
———контроль в нисходящем тестировании 86—87
HIPO 63—64, 72
— формализованное описание проекта 63—64
IBM Система/360 19, 36, 83, 132, 134, 142, 144, 206, 230, 251, 272, 314
— Система/370 36, 83, 123, 206, 251, 314

Каноническая форма проекта 51—52
Кобол, имитация CASE 181—182
— исключение GOTO 35
— использование ALTER 29, 43
— отсутствие блочной структуры 203, 215
— ошибки программирования 333, 341-345
— пакеты отладочных программ 161
— реализация таблиц решений 132—134
———функционального блока 178
— символьные параметры 142, 146
— средства реализации структурного программирования 180—181
Код ASCII, 150, 153, 161
— BCD 153, 161
— EBCDIC 153
Комментарии 22—23
— защита от ошибок 272—273
— неадекватность 23—24
— частота употребления 22—23
Константы (см. Символьные параметры)

Макрокоманды (см. Язык ассемблера)
Математическое обеспечение (см. Программы)
Моделирование 319—320
Модельная подпрограмма (см. Фиктивный модуль)
Модулей вертикальная организация 77
— горизонтальная организация 77
— независимость 119—120
— — от баз данных 119—120
————выбора алгоритма 119
————значений аргументов 119
————внутренних переменных 119
————— структуры управления 12*J
— тестирование (см. Тестирование модулей)
Модульное программирование 116—117
———использование подпрограмм 124-125
———отладка 327—328
— — отличие от структурного программирования 180
———связь со структурным программированием 116—117
———символьные параметры 139—150
———сопровождение 37—38
———таблицы решений 125—139
Модульность 121—123
— время ЦП 122—123
— в системах реального времени 123
— независимость модулей 119—120
— оперативная память 123
— программирование 121—122
— размеры модулей 117—119
Модуль, определение 284—285
— фиктивный (см. Фиктивный модуль)
Модуля размер 66—68, 110, 117—119, 124, 175

Надежность программных средств 285—286
Нисходящая схема (см. Нисходящее проектирование; Нисходящее кодирование; Нисходящее тестирование)
Нисходящее кодирование 73—78
———модификации 99—100
———определение 72—73
———организация модулей в листинге 77—78
— — связь с нисходящим тестированием 73
— программирование (см. Нисходящее проектирование; Нисходящее кодирование; Нисходящее тестирование)
— проектирование 53—54
———модификации 99—100
———пример 55—62
———проект IBM 109—110
———размеры модулей 66—68
———синонимы 51
———сочетание с восходящим 100—102
———стандартные подпрограммы 156—157
———структура данных 68—70
————уровни детализации 65—66
— — частные случаи 51—53
— — этапы 62—65
— тестирование 78—84, 302—307

———достоинства 85—91
————исключение тестирования системы 85—86
————наглядность результатов 88—89
————распределение времени 90
———логическое ядро программы 79, 81—82
———модификации 92—95, 102—104, 303—307
———пример 82—84
———проверка интерфейсов 85—87, . 306—307
———проект IBM 109—110
———сопоставление с восходящим 78-79
———сочетание с восходящим 93—95, 304
———условия испытаний 91, 305
•——— «челночные» проходы 93—95, 304

Общая память 30—31, 39, 154, 208
———в мультипрограммном режиме 29—30
———эффекты в отладке 347
ОС/360 19, 87, 172, 288, 294
Отладка 326—341
— бригадная 96
— динамическая (см. ДДТ)
— защита от ошибок 256—257, 326
— модульное программирование 326
— определение 287, 290
— организация программы 229, 232—233, 326
— проблемы сопровождения 37
— стандартные пакеты 161
— тестирование 325
Отладки методы в режиме онлайн (см. ДДТ)
———дампы 161, 345—347
———трассировка 161, 347—349
Ошибки ввода-вывода 270—271
— вероятность предупреждения 289
— виды 341—345
— выявление в нисходящем тестировании 86—87
— выявляемые тестированием 296—299
— интерфейсов 327, 328
— математического обеспечения 288—289
— математическое моделирование 320
— определение 286
— перфорации 341
— постоянство и повторяемость 331—333 ^
— при пользовании общей памятью 29—30
— проблемы сопровождения 36—39
— Система IBM/360 288—289

ПЛ/1, исключение GOTO 35
— средства реализации структурного программирования 180—191
— пакеты отладочных программ 161
— препроцессорные возможности 145—146
— реализация таблиц решений 131—132
— реализация функционального блока 178
— ошибки программирования 341—345
— эффективность 122
ПЛ/С 206
ПЛ/360 206
Повторное тестирование (см. Тестирование автоматизированное)
Подпрограммы ввода-вывода 160
— издержки применения 211—212
— насыщенность комментариями 22—23
— стандартные 155—162
*———библиотечные 155
Подсистема 285
Программ уровни сложности 290—296
———— гиперсложные 295—296
————сверхсложные 294—295
————сложные 292—294
————средней сложности 291—292
————простые 290—291
Программа ДЕНЬГИ 55—62, 385—389
Программирование избыточное 318—319
— модульное (см. Модульное программирование)
— нисходящее (см. Нисходящее проектирование; Нисходящее кодирование)
— стандарты и нормы 299, 309, 319
— структурное (см. Структурное программирование)
Программирования принципы 14—15
— производительность (см. Программисты производительность)
— языки (см. Фортран; Кобол; ПЛ/1; Алгол; Язык ассемблера)
Программиста «хорошего» качества 15—19
Программисты, производительность 110-112, 173—174
— различия в способностях 18—19
— сопровождения 36—37
— черты индивидуализма 23—25, 37—38

Программы 13, 285
— комментарии 22—25
— многократное обращение 232
— правильности доказательство 172—173, 28&-287
— сложность 13
— спецификации требований 13, 20
— структурированные (см. Структурное программирование)
Проект IBM 106—112, 168—169, 208« 210, 310
PDP-8 141, 152, 248
PDP-10 132, 135, 206, 251

Символьные параметры 139—150
———задание значений 146—150
———константы 142—146
Система 285
Систематическое программирование (см. Нисходящее проектирование)
Системные программы, 25, 205—207
Системный аналитик 68, 70—71
Снобол 176
Сопровождение 287—288
— программы без комментариев 22—23
— рекомендации 38—39
— стоимость 36
Стандартные подпрограммы 155—157
———недостатки 157—159
———примеры 159—162
Структурированный разбор 97
— — процедурные аспекты 97—98
———связь с тестированием 310—311
———участие руководителя 97
~———цели 96
Структурное программирование, исключение GOTO 176
———метод булевого признака 196—200
————дублирования кодов 186—192
-~ — основные конструкции 177—200
———отличие от модульного 180
———предпосылки и основания 166—170
———проблемы сопровождения 37—38
———расширения 181—184
———спецификации 72
———эффективность 175—176
Суперпрограммисты (см. также Программисты) 16—17
^- пример 24—25
— эксперименты IBM 106
CASE 181—182, 233, 276
— реализация 181—183

Таблиц решений метод 32
————достоинства 125
———пример 126—139
Таблицы решений, реализация 130—134
Тестирование 286
— автоматизированное 311—312
———генерация данных 312—314
———повторное 316—317
———тест-мониторы 317—318
— выявляемые ошибки 296—299
— затраты 21—22
— исчерпывающее 300—301
— математические модели надежности 320
— модулей 79, 300—301
— основной принцип 314
— подсистем 79, 300—301
— программных кодов 79, 300—301
— системы 79, 301—302
———выбор критериев 301
— сопоставление с верификацией 21
— этапы 299—307
Тестирования методы 318—320
— проблемы 288—290
Тестовая среда 91
Трассировка (см. Отладки методы)

Управление разработкой 21, 39—44
———гибкость программы 39—40
———оценка прогресса 21
———стоимость 40—42
Условия функционирования (см. Тестовая среда)
Устройства ввода-вывода 151

Файлы общего пользования 26
Фиктивный модуль 79
Фортран, использование GOTO 28, 32, 204
— нестандартное использование 28
— нецелесообразность исключения GOTO 35, 204
— отсутствие блочной структуры 204
———конструкции DO WHILE 204
————IF THEN ELSE 204
— ошибки программирования 333, 341—345
— пакеты отладочных программ 161
— препроцессоры 204
— реализация таблиц решений 130—132
———функционального блока 178
———CASE 181—183
— символьные параметры 140—142,146
— средства реализации структурного программирования 180—181

Эффективность программирования 43—44
———стандартные подпрограммы 155—157

Язык ассемблера^ макрокоманды 207
———ошибки программирования 341—345
———пакетыотладочных программ 161
———реализация таблиц решений 132—134
————функционального блока 178
———символьные параметры 139
— — сложные команды 250—252
Языки программирования, неоднозначность 308—309
<— системного программирования 205—207

ОГЛАВЛЕНИЕ

От редактора русского перевода 5
Предисловие 7

Глава 1. ОТЛИЧИТЕЛЬНЫЕ ОСОБЕННОСТИ «ХОРОШЕЙ» ПРОГРАММЫ ДЛЯ ЭВМ 13

1.0. Введение 14
1.1. Какими качествами обладает хороший программист? 15
1.2. Какими качествами должна обладать хорошая программа? . . . 19
1.3. Некоторые заключительные замечания относительно «качества» программ 44
Вопросы 45

Глава 2. НИСХОДЯЩЕЕ ПРОЕКТИРОВАНИЕ ПРОГРАММ 51

2.0. Введение 51
2.1. Нисходящее проектирование 53
2.2. Нисходящее кодирование 72
2.3. Нисходящее тестирование 78
2.4. Альтернативы, варианты и трудности нисходящего проектирования 99
2.5. Исследования и примеры. Разработка проекта фирмы IBM бригадным методом программирования 106
Литература 112
Вопросы 113

Глава 3. МОДУЛЬНОЕ ПРОГРАММИРОВАНИЕ 116

3.0. Введение 116
3.1. Определение модульности 117
3.2. Преимущества и недостатки модульности 120
3.3. Методы построения модульных программ 124
3.4. Стандартные подпрограммы 155
Литература 162
Вопросы 162

Глава 4. СТРУКТУРНОЕ ПРОГРАММИРОВАНИЕ 166

4.0. Введение 166
4.1. Основные предпосылки структурного программирования 166
4.2. Назначение и истоки структурного программирования 170
4.3. Теория и методы структурного программирования 176
4.4. Другие аспекты структурного программирования 207

4.5. Рассмотрение практических вопросов структурного программирования . . . 208
Литература . . . 216
Вопросы . . . 218

Глава 5. СТИЛЬ В ПРОГРАММИРОВАНИИ: ПРОСТОТА И ЯСНОСТЬ 228

5.0. Введение . . . 228
5.1. Обзор предложений по разработке простых программ . . . 230
5.2. Дополнительные методы повышения читабельности программ . . 243
Литература . . . 253
Вопросы . . . 254

Глава 6. ПРОГРАММИРОВАНИЕ С ЗАЩИТОЙ ОТ ОШИБОК . . . 256

6.0. Введение . . . 256
6.1. Возражения против программирования с защитой от ошибок , . 257
6.2. Что необходимо контролировать в программе? . . . 260
6.3. Методы программирования с защитой от ошибок . . . 267
Вопросы . . . 280

Глава 7. ПРИНЦИПЫ ТЕСТИРОВАНИЯ ПРОГРАММ . . . 283

7.0. Введение . . . 284
7.1. Понятия и определения . . . 284
7.2. Масштабы проблемы тестирования . . . 288
7.3. Уровни сложности тестирования . . . 290
7.4. Виды ошибок, которые должны выявляться при тестировании 296
7.5. Этапы тестирования . . . 299
7.6. Конструирование программ с целью облегчения тестирования 307
7.7. Автоматизированные способы тестирования . . . 311
7.8. Другие методы тестирования . . . 318
Литература . . . 321
Вопросы . . . 321

Глава 8. ПРИНЦИПЫ И СПОСОБЫ ОТЛАДКИ . . . 325

8.0. Введение . . . 325
8.1. Методологические и стратегические принципы отладки . . . 326
8.2. Типичные ошибки и погрешности программирования . . . 341
8.3. Классические способы и приемы отладки . . . 345
8.4. ДДТ-подсистемы динамической отладки . . . 349
8.5. Реализация простой версии ДДТ . . . 361
Литература . . . 375
Вопросы . . . 376

ПРИЛОЖЕНИЯ. УПРАЖНЕНИЯ И ЗАДАЧИ ДЛЯ КЛАССНЫХ ЗАНЯТИЙ . . . 379

Введение . . . 379
А. Денежная задача . . . 383
Б. Задача фирмы «Независимый удобритель» . . . 390
В. Задача модификации главного файла . . . 403
Г. Программа для игры в тик-так . . . 408

ПРЕДМЕТНЫЙ УКАЗАТЕЛЬ . . . 409

УВАЖАЕМЫЙ ЧИТАТЕЛЬ!

Ваши замечания о содержании книги, ее оформлении, качестве перевода и другие просим присылать по адресу: 129820, Москва, И-110, ГСП, 1-й Рижский пер., д. 2, издательство «Мир»,

Э. Йодан

СТРУКТУРНОЕ ПРОЕКТИРОВАНИЕ
И КОНСТРУИРОВАНИЕ ПРОГРАММ

Ст. научн. редактор М. Б. Великовский
Мл. научн. редактор М. В. Архипова
Художник А. В. Шипов
Худож. редактор Л. Е. Безрученков
Технический редактор Т. Н. Полюшкина
Корректор Л. Д. Панова

ИБ № 1068

Сдано в набор 05.05.78.Подписанокпечати05.10.78. Формат 60x90/16. Бумагатипографская№2. Латинскаягарнитура. Высокаяпечать. Объем; 13,00 бум.л., 26,00усл.печ.л., Уч.-изд. л. 27,45. Изд.№20/9437. Тираж 27 000 экз. Зак. 2669. Цена 2 р. 20 к.
Издательство «Мир»
Москва, 1-й Рижский пер., 2

Ордена Октябрьской Революции
и ордена Трудового Красного Знамени
Первая Образцовая типография имени А. А. Жданова
Союзполиграфпрома при Государственном комитете
СССР по делам издательств,
полиграфии и книжной торговли,
Москва, М-S4, Валовая, 28